WARNING!!!

THIS IS A NUTS-AND-BOLTS TALK!
Goals

- Give a tour of Gambit Scheme implementation
- Programmer’s perspective -- how to use it!
- Implementation of system -- how it works!
Talk Overview

- Brief overview of Scheme and Gambit
- Compiler and portability
- Highlights of Gambit Scheme language and implementation
- Applications and demos
Scheme and Gambit Overview
1975: Sussman & Steele design Scheme at MIT

Few but powerful building blocks

Small language... “Do it yourself” philosophy

Scheme is a “Lisp-1”: unified name space for functions, macros and variables

1978: RABBIT Scheme compiler thesis: reduce complex constructs to a small core based on the lambda calculus => simple and efficient compiler
Evolution of Standards

- "Academic era": concerns for purity

- "Real-world era": practical concerns
  - Scheme Request for Implementation (SRFI), over 100 documents, ongoing since 1998
  - Evolution by revolution: R6RS (2007) => 160 page spec, controversial, R7RS (?)
Over 50 implementations of Scheme, many toys and over 15 mature systems:

- **Compilers to VM and interpreters**: Gauche, Guile, Kawa, Scheme48, SCM, SISC, Ypsilon

- **Compilers to native code (including JIT)**: Chez Scheme, Ikarus, Larceny, MIT Scheme, MzScheme, Racket (PLT Scheme)

- **Compilers to C**: Bigloo, Chicken, Gambit-C, Stalin, Petit Larceny
Gambit System Evolution

- 1989: Compiler to M68K, no interpreter, no GC
- 1991: MacGambit
- 1993: Message passing implementation of futures on 90 processor BBN Butterfly
- 1994: C back-end, first commercial use
- 2004: Gambit v4, threads, I/O, LGPL/Apache
Gambit Uses in Academia

- **Education**: PL concepts, compilers, AI, math (numerical analysis, homework on web)

- **Research**: concurrent systems, real-time GC, continuations, optimizations, FPGAs, ...

- **Compiler research**:
  - Scheme as UNCOL: Erlang/Java/JavaScript
  - Scheme -> C/JavaScript/VHDL
  - For embedded sys: BIT, PICBIT, PICOBIT (< 20 kB R4RS on microcontrollers)
Gambit Commercial Uses

- Selling Point: product configuration (Gambit used as back-end for custom OO language)
- EdScheme: a Scheme for teaching math
- Parallel Geometry: CAD for exact 3D modeling
- Quantz: casual video game (PC/Mac/Linux)
- iPhone games (Farmageddon, Reverso)
- JazzScheme: an OO Scheme, with many libraries, GUI, IDE (PC/Mac/Linux)
- Auphelia inc: Enterprise Resource Planning
Gambit-C Goals

- A Scheme system that is
  - conformant to R5RS and robust (no bugs)
  - portable
  - efficient (i.e. fast)

- Provide simple building blocks for
  - developing practical applications
  - building more complex languages

- Avoid “being in the programmer’s way”
Gambit has 2 main programs
- **gsi**: interpreter (best for debugging but not fast)
- **gsc**: compiler (which includes interpreter)

Interpreted and compiled code can be freely mixed

```
% gsi
Gambit v4.6.0
> (load "fib")
55
"/Users/feeley/fib.scm"
> (fib 20)
6765
> (exit)
```

```
% gsi fib.scm
55
% gsc fib.scm
% gsi fib.o1
55
% gsc -exe fib.scm
% ./fib
55
% gsc -c fib.scm
```
Compiler and Portability
Portability

- **gsc** generates C code that is **independent** of the target processor, C compiler and OS
- Code is compilable by any C or C++ compiler, on 32/64 bit processors, any endianness
- *Trampolines* are used for supporting tail calls (Scheme stack managed separately from C’s)
Gambit Virtual Machine

- GVM is the compiler’s intermediate language
- Register based VM (nb of regs depends on BE)
- First few parameters in registers, rest on stack
- Stack is allocated implicitly (no push/pop)
- No call instruction, only jump
- jump/poll instruction indicates safe points where interrupts are allowed and where stack and heap overflows are checked
C Back-End

mod1.scm

(print
(max 11 22))

non-tail-call

tail-call

mod1.gvm

#1 fs=0 entry-point 0 ()
   STK1 = R0
   R2 = '22
   R1 = '11
   R0 = #2
   jump/poll fs=4 max 2

#2 fs=4 return-point
   R0 = STK1
   jump/poll fs=0 print 1

mod1.c

#include "gambit.h"

BEGIN_SW
DEF_SLBL(0,L0_mod1)
   SET_STK(1,R0)
   SET_R2(FIX(22L))
   SET_R1(FIX(11L))
   SET_R0(LBL(2))
   ADJFP(4)
   POLL(1)
DEF_SLBL(1,L1_mod1)
   JUMPGLO(NARGS(2),
   1,G_max)
DEF_SLBL(2,L2_mod1)
   SET_R0(STK(-3))
   ...

Note: GVM and C code modified for readability
System Portability

- **gambit.h** allows late binding of GVM implem.
- a **configure** script tunes the **gambit.h** macro definitions to take into account:
  - target OS, C compiler, pointer width, etc
- E.g. trampoline operation **BEGIN_SW** becomes
  - “switch (pc-start) ...” by default
  - “goto *(pc->lbl);” if gcc is used
Gambit adopts a Scheme-in-Scheme approach
primitives, interpreter, debugger, bignums, ...

Non-Scheme code (~ 30%) is mainly for OS interface and is in portable C (no asm code!)

Runtime relies only on standard C libraries

Compiled application can be distributed as the set of generated “.c” files (Gambit not needed on the target system, great for embedded sys)
System Portability

- configure
  - config.h
  - gambit.h
  - main.c
  - os.c
  - mem.c
- runtime library
  - _kernel.scm
  - _num.scm
  - _io.scm
- application
  - app.scm
- gsc
  - _kernel.c
  - _num.c
  - _io.c
  - app.c
  - app_.c
- CC
  - app.exe
- link file
- runtime library
- application
- gsc
- CC
- app.exe
- link file
System Portability

- Compiles “out-of-the-box” for Intel, SPARC, PPC, MIPS, ARM, etc
- Porting to a new processor: 0 to 60 minutes
- Unusual porting examples:
  - Nintendo DS (ARM, 4 MB RAM)
  - Linksys WRT54GL (MIPS, 16 MB RAM)
  - iPhone/iTouch (ARM, 128 MB RAM)
  - Xilinx FPGA (PPC, few MB RAM, no OS)
Gambit Scheme Language
Main Extensions

- Declarations
- Namespaces
- Threads, I/O, Serialization
- Scheme Infix eXtension (SIX)
- Foreign Function Interface (FFI)
Declarations
Declarations

- By default Gambit obeys R5RS semantics
- This has an impact on performance
- Declarations allow the programmer to indicate where it is OK to make some assumptions, which enable various optimizations

```
(car x) ;; 1) read the “car” global variable
        ;; 2) check that it is a function
        ;; 3) call the function
```

```
(declare (standard-bindings))
(car x) ;; car is known to contain the car
        ;; function so the compiler can inline it
```
### Other Declarations

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(block)</code></td>
<td>Assume global vars defined in this file are not mutated outside it</td>
</tr>
<tr>
<td><code>(fixnum)</code></td>
<td>Fixnum arithmetic only</td>
</tr>
<tr>
<td><code>(flonum)</code></td>
<td>Flonum arithmetic only</td>
</tr>
<tr>
<td><code>(not safe)</code></td>
<td>Assume no type checks fail</td>
</tr>
<tr>
<td><code>(debug)</code></td>
<td>Generate debug info</td>
</tr>
<tr>
<td><code>(not proper-tail-calls)</code></td>
<td>Turn off TCO</td>
</tr>
</tbody>
</table>
Impact on Performance

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1))
          (fib (- n 2))))
)
(fib 40)
```

MacBook Pro
2.8 GHz Intel Core 2 Duo
4 GB RAM

<table>
<thead>
<tr>
<th>Description</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>no declaration (i.e. pure R5RS semantics)</td>
<td>5.68</td>
</tr>
<tr>
<td>(declare (standard-bindings))</td>
<td>4.80</td>
</tr>
<tr>
<td>+ (declare (block))</td>
<td>3.30</td>
</tr>
<tr>
<td>+ (declare (fixnum))</td>
<td>2.70</td>
</tr>
<tr>
<td>+ (declare (not safe))</td>
<td>1.11</td>
</tr>
<tr>
<td>gcc -O2 fib40.c</td>
<td>1.63</td>
</tr>
<tr>
<td>sbcl &lt; fib40.lisp (no declaration)</td>
<td>4.24</td>
</tr>
</tbody>
</table>
Main Optimizations

- Inlining
  - Primitive functions (\texttt{car, cons, map, ...})
  - User functions, including recursive functions
  - Speculative inlining of primitive functions (when binding of global var unknown)
- Lambda lifting
- Copy/constant propagation, constant folding
Namespaces
Namespaces

- Namespace declarations allow mapping identifiers to spaces
- They are lexically scoped
- They work by prefixing *unqualified* identifiers into *qualified identifiers* of the form `space#id`

```
(##namespace ("foo#"))  ID -> foo#ID (for all ID)
(##namespace ("bar#" a b))  a -> bar#a  b -> bar#b
```
Namespaces as Modules

- Can be used as a simple module system

```
stk#.scm

(define (empty) '())
(define (push x s) (cons x s))
(define (pop s) (cdr s))
(define (test)
  (if (equal? (push 1 (empty)) '(1))
      "good!"
      "bad!")

```

```
stk.scm

(define (empty) '())
(define (push x s) (cons x s))
(define (pop s) (cdr s))
(define (test)
  (if (equal? (push 1 (empty)) '(1))
      "good!"
      "bad!")

```

```
~lib/gambit#.scm

(define (empty) '())
(define (push x s) (cons x s))
(define (pop s) (cdr s))
(define (test)
  (if (equal? (push 1 (empty)) '(1))
      "good!"
      "bad!")
```

```
~lib/gambit#.scm

(define (empty) '())
(define (push x s) (cons x s))
(define (pop s) (cdr s))
(define (test)
  (if (equal? (push 1 (empty)) '(1))
      "good!"
      "bad!")

```
Namespaces as Modules

```
client.scm
(#include "stk#.scm")
(define (test)
  (pp (pop (empty)))))
```

```
(define (test)
  (pp (stk#pop (stk#empty)))))
```
Namespaces as Modules

Quiz: Why “##” prefix?

```
(stk#.scm)

(##namespace
  ("stk#" empty push pop))

(##include "~~lib/gambit#.scm")

(define (empty) '())
(define (push x s) (cons x s))
(define (pop s) (cdr s))

(define (test)
  (if (equal? (push 1 (empty)) (1))
      "good!"
      "bad!")

(##namespace
  ("" cons car cdr define ...))
```

```
(stk.scm)

(##namespace ("stk"))
(##include "~~lib/gambit#.scm")
(##include "stk#.scm")

(define (empty) '())
(define (push x s) (cons x s))
(define (pop s) (cdr s))

(define (test)
  (if (equal? (push 1 (empty)) (1))
      "good!"
      "bad!")
```
Threads, I/O, Serialization
Threads

- Green threads
- Preemptive scheduler with priorities
- Very lightweight and scalable
- Thread = descriptor (324 bytes) + continuation
- Thread creation/synchronization ~ 0.5 \( \mu s \)
- \( O(\log N) \) enqueue/dequeue operations
- Supports millions of active threads (in ~ 1GB)
Threads: API

- API of SRFI-21 “Real-time multithreading”
- Objects: threads, mutexes, condition variables
- Priority inheritance

```
(define n 0)
(define m (make-mutex))

(define (increment)
  (do ((i 100000000 (- i 1))) ((= i 0))
      (mutex-lock! m)
      (set! n (+ n 1))
      (mutex-unlock! m)))

(define threads (list (make-thread increment)
                      (make-thread increment)))

(for-each thread-start! threads)
(for-each thread-join! threads)
(print n) => 2000000000
```
Threads: Scheduler

- Scheduler is implemented in Scheme
  - Suspension: done internally with call/cc
  - Preemption: done with heartbeat interrupts

- Threads have
  - a continuation
  - a priority level and a quantum
  - a “specific” field (for thread local storage)
  - a mailbox (for thread-send/receive)
Threads: Mailboxes

- Mailboxes simplify thread interaction
- A mailbox acts as an operation serializer

```scheme
(define (make-server op)
  (thread-start!
    (make-thread
      (lambda ()
        (let loop ()
          (let ((msg (thread-receive)))
            (thread-send (car msg) ;; client
             (op (cdr msg)))
              (loop))))))

(define file-server (make-server get-file))

(thread-send file-server
  (cons (current-thread) 
    "/etc/passwd"))

(print (thread-receive)) ;; print file
```
I/O

- I/O is compatible with R5RS text-only model
- Extensions:
  - control over character and EOL encoding
  - binary I/O
  - bulk I/O
  - nonblocking I/O (on all port types)
- Port types: file, directory, OS process, TCP client, TCP server, string, vector, pipe, ...
I/O: Port Types

- Port types are organized in a class hierarchy
- Byte port <: character port <: object port

read/write  read-char/write-char  read-u8/write-u8

readtable  char encoding

byte port
char port
obj port
The procedures which create ports allow a settings list specifying the set of parameters.

```scheme
(define (log-msg msg)
  (with-output-to-file
    (list path: "~/log"
         append: #t
eol-encoding: 'cr-lf
char-encoding: 'UTF-8
output-width: 80)
    (lambda ()
      (println msg))))

(log-msg "hello")
(log-msg "world!")
```
I/O: Directory Ports

- Directory ports allow constant space iteration over directories
- Reading a directory port yields the next entry (directory ports are thus not character ports)

```scheme
(define (for-each-directory-entry dir proc)
  (let ((dir-port (open-directory dir)))
    (let loop ()
      (let ((file (read dir-port)))
        (if (eof-object? file)
            (close-port dir-port)
            (begin
              (proc file)
              (loop)))))))

(for-each-directory-entry "~" println)
```
I/O: TCP Ports

- 2 kinds: TCP server ports and TCP client ports
- Reading a TCP server port yields a port which is the accepted connection with the client (TCP server ports are thus not character ports)

```
(define (start-server port-num proc)
  (let ((serv-sock (open-tcp-server port-num)))
    (let loop ()
      (proc (read serv-sock))
      (loop))))

(start-server 8080
  (lambda (conn)
    (display "hello\n" conn)
    (close-port conn)))

... (open-tcp-client "localhost:8080") ...
```
I/O: String Port Generalization

- Generalized to objects, characters and bytes
- *Pipe ports*: port pairs connected by a FIFO
- Useful for interthread communication

```scheme
(define a
  (open-output-string))
(write (list 1 2 3) a)
(get-output-string a)
  => "(1 2 3)"
(define b
  (open-output-vector))
(write 1 b) (write 2 b)
(get-output-vector b)
  => #(1 2)

(call-with-values
  (lambda ()
    (open-string-pipe))
  (lambda (i o)
    (write 1 o) (newline o)
    (write 2 o) (newline o)
    (force-output o)
    (println (read i)) ;; 1
    (println (read i))) ;; 2
```

`or vector`
I/O: Nonblocking I/O

- Ports have a timeout for input and output ops, which defaults to infinity, i.e. blocking op
- This can be set for all types of ports including TCP server and directory ports

```
(define (rl-with-timeout timeout)
  (let* ((port (current-input-port))
         (line (call/cc
                (lambda (abort)
                  (input-port-timeout-set! port timeout
                                           (lambda () (abort #f)))
                  (read-line port)))))
    (input-port-timeout-set! port +inf.0) line))

(rl-with-timeout 10) ;; #f if no input after 10 secs
```
Objects can be serialized into byte vectors
- Supports closures, continuations, cycles
- Useful for distributed computing (Termite)
- Source and destination can be of different type (processor, OS, word width, endianness, ...)

Serialization
Serialization

- Example: parallel processing

```scheme
;; server running on machines foo and bar
(let ((serv-sock (open-tcp-server "+:5000")))
  (let loop ()
    (let ((conn (read serv-sock)))
      (write (object->u8vector
               ((u8vector->object (read conn)))
               conn)
             (close-port conn)
             (loop))))
```
;; client somewhere on the network

(define (on address thunk)
  (thread-start!
    (make-thread
     (lambda ()
      (let ((conn (open-tcp-client address)))
        (write (object->u8vector thunk) conn)
        (force-output conn)
        (let ((result (u8vector->object (read conn))))
          (close-port conn)
          result)))))))

(define (test n)
  (define (f n)
    (if (< n 2) 1 (* n (f (- n 1))))
    (let ((a (on "foo:5000" (lambda () (f (+ n 1)))))
          (b (on "bar:5000" (lambda () (f n))))
          (/ (thread-join! a) (thread-join! b))))
  (test 1000) => 1001)
Serialization

- Extra “encoder/decoder” parameter allows custom encoding, which is useful for otherwise unserializable objects (ports, threads, ...)

```scheme
(define (print-to port)
  (lambda (x) (display x port)))

(define a (print-to (current-output-port)))

(define cop-repr 'the-cop) ;; todo: unique record

(define (encoder x)
  (if (eq? x (current-output-port)) cop-repr x))

(define (decoder x)
  (if (eq? x cop-repr) (current-output-port) x))

(define b (object->u8vector a encoder))
(define c (u8vector->object b decoder))

(c "hello") ;; prints to current-output-port
```
Scheme Infix eXtension
SIX: Goals

- Infix syntax close to C/Java

- Multiple goals:
  - Reduce adoption barrier for non-Lispers (emphasize Scheme semantics, not syntax!)
  - Compact notation for arithmetic expressions
  - Built-in parser for compiler course
SIX: "\" Escapes

- Idea: a "\" switches between prefix and infix

- In prefix syntax: \ <infix statement>

- In infix syntax: \ <prefix expression>

```scheme
(let (((v '#(10 17 44))
  (s 0))
  \ for (int i=0; i<\vector-length(v); i++)
  s += v[i]*v[i];
  (println s))
```
SIX: Syntax

- SIX extends C’s syntax with anonymous and nested functions, list constructor, ({{...}}) blocks, Prolog clauses, etc

```c
\ obj foo(int n)
 { int fact(int x)
   { if (x<2) 1; else x*fact(x-1); }  

   map(int (int s) { fact(n<<s); }, 
       [2, 3, 4, 5]);
  }

(\ foo 1) => (24
  40320
  20922789888000
  263130836933693530167218012160000000
)```
SIX: Semantics

- Reader builds AST as a S-expression
- Semantics is given by predefined "six.XXX" macros, which can be redefined

```
> ' \ "hello " + "world!";

(six.x+y (six.literal "hello ")
  (six.literal "world!"))

> (define-macro (six.x+y x y)
  `(string-append ,x ,y))

> \ "hello " + "world!";

"hello world!"
```
Foreign Function Interface
**FFI**

- Allows calls between Scheme and C (both ways)
- Useful for linking with C libraries
- Automatic representation conversions:

<table>
<thead>
<tr>
<th>C</th>
<th>Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int</code></td>
<td><code>int</code></td>
</tr>
<tr>
<td><code>unsigned int</code></td>
<td><code>unsigned-int</code></td>
</tr>
<tr>
<td><code>char</code></td>
<td><code>char</code></td>
</tr>
<tr>
<td><code>char *</code></td>
<td><code>char-string</code></td>
</tr>
<tr>
<td></td>
<td><code>nonnull-char-string</code></td>
</tr>
<tr>
<td></td>
<td><code>UTF-8-string</code></td>
</tr>
<tr>
<td><code>T *</code></td>
<td><code>(pointer T [(type-id...) [release-fn]])</code></td>
</tr>
<tr>
<td></td>
<td><code>nonnull-pointer T ...</code></td>
</tr>
</tbody>
</table>
c-define-type gives names to foreign types

(c-define-type boolean int) ;; type alias
(c-define-type Window "Window") ;; new type
(c-define-type Window* (pointer Window (Window*) "release_Window"))

;; GC and (foreign-release! ptr) call release_fn

;; a type with custom conversion functions:
(c-define-type foo "foo" "foo_c2s" "foo_s2c")
FFI: Calling C

- \textit{c-lambda} yields a Scheme proxy of C function

\begin{verbatim}
(c-declare "#include <stdio.h>")

(c-define-type FILE "FILE")
(c-define-type FILE* (pointer FILE))

(define fopen
  (c-lambda (char-string char-string) FILE* "fopen"))

(define fgetc
  (c-lambda (FILE*) int "fgetc"))

(define fputc
  (c-lambda (int FILE*) int "fputc"))

(define fclose
  (c-lambda (FILE*) int "fclose"))
\end{verbatim}
FFI: Calling Scheme

- **c-define** defines a function callable from C

```scheme
;; hook into Scheme’s eval from C:
(c-define (eval-string str)
    (char-string) char-string "eval_string" ""
    (object->string
       (eval (with-input-from-string str read))))
```
Other Extensions
Tables

- Hash-tables with several options including:
  - Test: `eq?`, `equal?`, ...
  - Hash function: `eq?-hash`, `equal?-hash`, ...
  - Load factor limits (low and high)
  - Key and value reference “weakness”

```scheme
(define t (make-table test: eq?
      weak-keys: #t))

(define obj (cons 1 2))

(table-set! t obj 99)
(table-ref t obj) => 99

(set! obj #f) ;; GC will remove entry from t
```
Wills

- Will objects control object finalization

```
(define obj (cons 1 2))
(make-will obj (lambda (x)
    (pp x)
    (finalize x)))
(set! obj #f) ;; GC will call action procedure
```
Serial Numbers

- Serial numbers are used by the printer to identify objects which can’t be read
- Convenient for debugging

```
> (let ((n 2)) (lambda (x) (* x n)))
#<procedure #2>
> (pp #2)
(lambda (x) (* x n))
> (map #2 '(1 2 3 4 5))
(2 4 6 8 10)
> ,(v #2)
1> ,e
n = 2
1> (set! n 10)
1> ,t
> (map #2 '(1 2 3 4 5))
(10 20 30 40 50)
```
Records

- Extensible records (using single inheritance)
- Serializable
- Field attributes

```scheme
(define-type pt x y) ;; (make-pt x y)
;; (pt? obj)
;; (pt-x obj)
;; (pt-x-set! obj val) ...

(define-type person
id: B3D36093-BC54-7D78E7CB7ADA
extender: define-type-of-person
(name read-only:))

(define-type-of-person employee
id: C4DA4307-A1A1-E7F7461E8DDF
(employer unprintable: equality-skip:))
```
Homogeneous Vectors

- Vectors of fixed width integers and floats

```scheme
(define v (make-f64vector 10 3.1416))
(f64vector-set! v 0 (* 2 (f64vector-ref v 0)))

;; u8vector    unsigned integers
;; u16vector
;; u32vector
;; u64vector

;; s8vector    signed integers
;; s16vector
;; s32vector
;; s64vector

;; f32vector   floating point numbers
;; f64vector
```
Optional/Named Parameters

- Similar to Common-Lisp
- Optional parameters, by position
- Named parameters use keyword objects

```
(define (fmt n
    #!optional (base 10)
    #!key (port (current-output-port)))
  (display (number->string n base) port))

(fmt 123)
(fmt 123 2)
(fmt 123 2 port: (current-error-port))
```
Memory Management
For portability, all memory allocated with `malloc`.

Small objects and cont. frames are MOVABLE.

Objects that are large or allocated by FFI are STILL.
MOVABLE Objects

- Allocation = pointer increment (HP or SP)
- Stop-and-copy compacting GC
- MOVABLE sections added/removed to maintain a given live ratio at end of GC (0.5 by default)

Diagrams:
- MOVABLE sections
- Small objects
- STILL objects (large or FFI)
- HP
- HL
- SL
- SP
- rc
- External reference (from C)
- Continuation frames
- 0.5 MB
STILL Objects

- Reference count for external refs simplifies FFI
- Mark-sweep compacting GC
- Reclaim when $rc=0$ and no refs from heap
PERMANENT Objects

- Not reclaimed or scanned by GC, C “static”
- Constant objects in Scheme program
- Descriptors of code points in Scheme program
  - function entry points
  - function call return points

### Scheme Code
```
(define f
  (lambda (x)
    (+ (g x) x)))
```

### C Code
```
void H_mod1(...)
{
  ...
}
```

### Memory Layout
```
<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>host:</td>
</tr>
<tr>
<td></td>
<td>nbp:1</td>
</tr>
<tr>
<td></td>
<td>nbc:0</td>
</tr>
<tr>
<td>2</td>
<td>host:</td>
</tr>
<tr>
<td></td>
<td>fs:4</td>
</tr>
<tr>
<td></td>
<td>map:0110</td>
</tr>
</tbody>
</table>
```
Continuations

- Continuation frames are “pushed” by moving SP.
- Typically, frames are “popped” on function return.
- `call/cc` protects captured frames with: \( SC := SP \).
- Protected frames are copied to TOS, never popped.
- Interrupt: \( SL := SC \).
(define (A) (B) (C) 1)
(define (B) 2)
(define (C) (call/cc D) 3)
(define (D k) (k 4))
Continuations

(define (A) (B) (C) 1)
(define (B) 2)
(define (C) (call/cc D) 3)
(define (D k) (k 4))
Continuations

\[
\begin{align*}
&\text{(define (A) (B) (C) 1)} \\
&\text{(define (B) 2)} \\
&\text{(define (C) (call/cc D) 3)} \\
&\text{(define (D k) (k 4))}
\end{align*}
\]
(define (A) (B) (C) 1)
(define (B) 2)
(define (C) (call/cc D) 3)
(define (D k) (k 4))
(define (A) (B) (C) 1)
(define (B) 2)
(define (C) (call/cc D) 3)
(define (D k) (k 4))
Continuations

(define (A) (B) (C) 1)
(define (B) 2)
(define (C) (call/cc D) 3)
(define (D k) (k 4))
Continuations

(define (A) (B) (C) 1)
(define (B) 2)
(define (C) (call/cc D) 3)
(define (D k) (k 4))
Continuations

- Live frames are copied to heap by GC (explicit links are added to form a chain of frames)
- Space for link reserved when frame is created
- An “underflow handler” is used to copy the next frame when SP = SC
- No overhead when `call/cc` not called
- Constant time `call/cc`
- Note: interleaving of frames from different threads
Third Party Stuff
Third Party

- Code repository: Gambit Dumping Grounds
- Libraries: OpenGL, MySQL, HTTP servers, Scheme to JS compiler, lexer and LALR parser generator
- **Black hole** module system
- **JazzScheme** system (OO extension + IDE + libs)
- **statprof** statistical profiler
Demos
iPhone Apps

Farmageddon

Play

Sound: On
Scores
Emacs Debugging

Gambit v4.6.0

> (load "fib.scm")
"/Users/feeley/fib.scm"
> (break fib)
> (fib 4)
*** STOPPED IN fib. "fib.scm"@2.8
1> .c
*** STOPPED IN fib. "fib.scm"@2.8
1> .c
*** STOPPED IN fib. "fib.scm"@2.8
1> .s
| > <
| #<procedure #2<br
*** STOPPED IN fib. "fib.scm"@2.10
1> .s
| > n
| 2
*** STOPPED IN fib. "fib.scm"@2.12
1> .s
| > 2
| 2
*** STOPPED IN fib. "fib.scm"@2.7
1> []

1:<-- *scheme*  All L24  (Inferior Scheme:run)--11:54PM  1.59--

(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2))))

:--- fib.scm  All L2  (Scheme)--11:54PM 1.59--
Jedi: Jazz/Gambit IDE
### Emilie: DB Front-End

**Bénéficiaire**

- **No dossier**: 4898
- **Dossier**: fermé
- **Ouverture**: 12/07/2006
- **Téléphone**: [storage]
- **Poste**: [storage]
- **Nom**: [storage]
- **Prénom**: [storage]
- **Conjoint**: [storage]
- **Conjoint poste**: [storage]
- **Adresse**: [storage]
- **Code postal**: [storage]
- **Ville**: [storage]
- **Naissance**: 07/06/19
- **Statut**: M
- **Sexe**: F
- **Type**: [storage]
- **Code BS**: [storage]
- **Agent BS**: [storage]
- **Raison**: agée
- **Fermeture**: 03/08/20

**Accompagnements**

<table>
<thead>
<tr>
<th>Date</th>
<th>Chauffeur</th>
<th>Km</th>
<th>Payeur</th>
<th>Endroit</th>
<th>Reçu</th>
<th>Notes</th>
<th>Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>26/09/2006</td>
<td>Camionnette 03</td>
<td>22</td>
<td></td>
<td>CLSC</td>
<td>10.00</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>12/09/2006</td>
<td>René</td>
<td>22</td>
<td></td>
<td>CLSC</td>
<td>10.00</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>29/09/2006</td>
<td>Jean-Louis</td>
<td>25</td>
<td></td>
<td>CLSC-PRÉLÈVEMENTS</td>
<td>10.00</td>
<td>0.00</td>
<td></td>
</tr>
</tbody>
</table>

**Services**

- **Accompagnement**: ✓
- **Popote**: □
- **Visite**: □
- **C.C.**: □
- **Point Rouge**: □
- **PAIR**: □
- **Sécurité**: □

**Centre Communautaire**

- **Aidants naturels**: [storage]
- **Café rencontre**: [storage]
- **Danse**: [storage]
- **Peinture**: [storage]
- **PAP**: [storage]
- **PIED**: [storage]
- **Scrabble**: □
- **Tai Chi**: □
- **Sorties**: [storage]

**Popote**

- **Étiquette**: □
- **Reçu**: □
- **Route**: [storage]
- **Route MJ**: [storage]
- **Mémo**: [storage]

**Cotisation**

- **Date renouvellement**: 01/08/2007
- **Date paiement**: 11/07/2006
- **Montant**: 15
- **Paiement**: env. 30-07-07

**Remarques**

- **Répondant**: Dalco [storage]
- **Info accompagnement**: [storage]
Interested?

Google “Gambit Scheme”

- Source and binary distributions
- Gambit wiki
- Gambit mailing list

Many thanks to:
- Guillaume Cartier (JazzScheme)
- Robert Lizee (Quantz)
- James Long (Farmageddon)