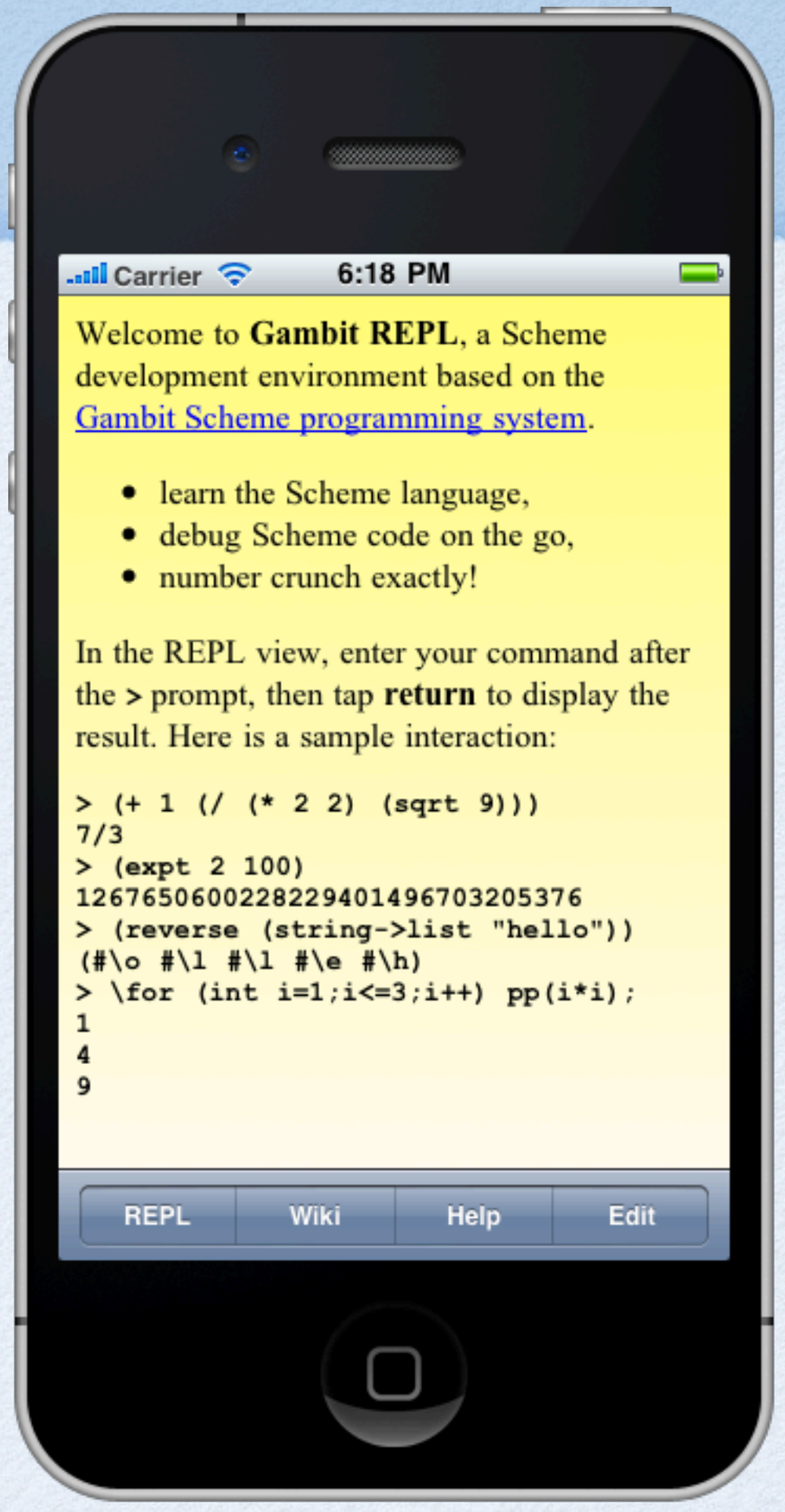
Gambit REPL

Marc Feeley

November 24, 2011

Overview

- Scheme
- Gambit
- Gambit REPL app
  - User interface
  - Development
  - Implementation



Scheme

Scheme

- 1975: Sussman & Steele design Scheme at MIT
- Few but powerful building blocks
  - simple uniform syntax (parenthesized prefix)
  - dynamically typed
  - functional and imperative programming
  - macros, closures, first-class continuations, tail-calls, garbage collection, ...
- Used by many institutions to teach CS

Scheme Example

```
(define (join words separator)
   (apply string-append
          (map (lambda (str) ;; a closure
                 (string-append separator str))
               words)))

 (define (path . parts) ;; a variadic function
   (join parts "/"))

(path "usr" "local" "bin") ⇒ "/usr/local/bin"

(define-macro (push val var) ;; a procedural macro
  `(set! ,var (cons ,val ,var)))

(define stack '())

(push 11 stack)
(push 22 stack)

stack ⇒ (22 11)
```

Evolution of Standards

- "Academic era": *concerns for purity*
  - Evolution by unanimous consent:
    R1RS (1978), R2RS (1985), R3RS (1986),
    R4RS (1991), R5RS (1998) => 50 page spec
- "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 (soon!)

Scheme Systems

- Over 50 implementations of Scheme, many toys and over 15 mature systems!
- Diverse implementation approaches:
  - Interpreters and VMs – ***Guile, Kawa, ...***
  - JIT compilers – ***Racket, Larceny, Chez, ...***
  - Compilers to C – ***Gambit, Bigloo, Chicken, ...***

Gambit

Gambit System Evolution

- 1989: Compiler to M68K, no interpreter, no GC
- 1991: MacGambit – compiler/interpreter/IDE
- 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
- 2011: Gambit REPL - interpreter for iOS

Gambit Goals

- A Scheme system that is
  - conformant to R5RS and robust (no bugs)
  - portable
  - efficient (i.e. fast)
  - embeddable
- Provide simple building blocks for
  - developing practical applications
  - building more complex languages
- Avoid “being in the programmer’s way”

GSI and GSC

- On workstations, 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
```

Portability

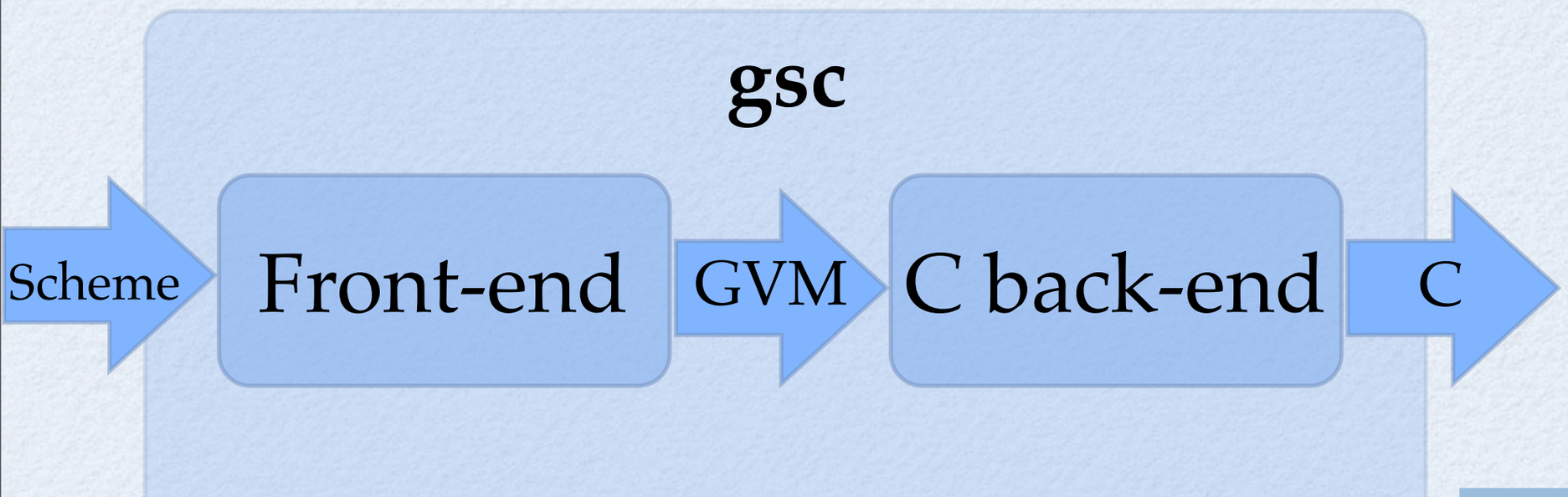
- **gsc** generates C code that is <u>independent</u> of the target processor, C compiler and OS
- Compilable by any C, C++, or ObjC compiler, on 32/64 bit processors, any endianness
- *Trampolines* are used for supporting tail calls (Scheme stack managed separately from C's)

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

Note: GVM and C code modified for readability



mod1.scm

```
(print
  (max 11 22))
```

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
```

non-tail-call

tail-call

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))
    ...
```

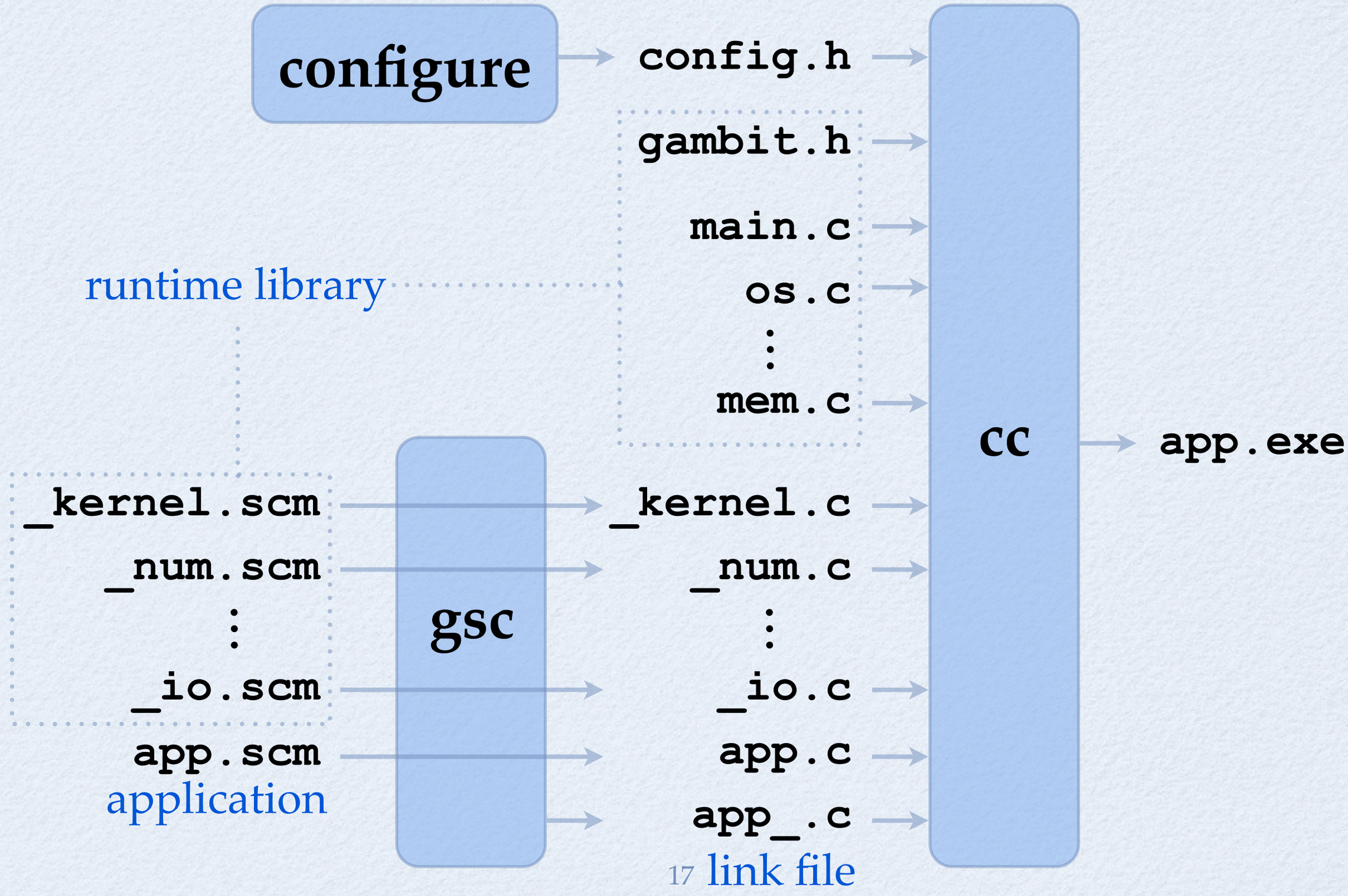
System Portability

- **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 using **gcc** (faster!)

System Portability

- 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



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/iPad (ARM, 128 MB RAM)
  - Xilinx FPGA (PPC, few MB RAM, no OS)

Main Extensions

- Declarations (to deviate from std semantics)
- Namespaces
- Green threads, mutex/cond.var., mailbox
- I/O – TCP, subprocesses, serialization, ...
- Hash tables and wills
- Foreign Function Interface (FFI)

Gambit REPL app

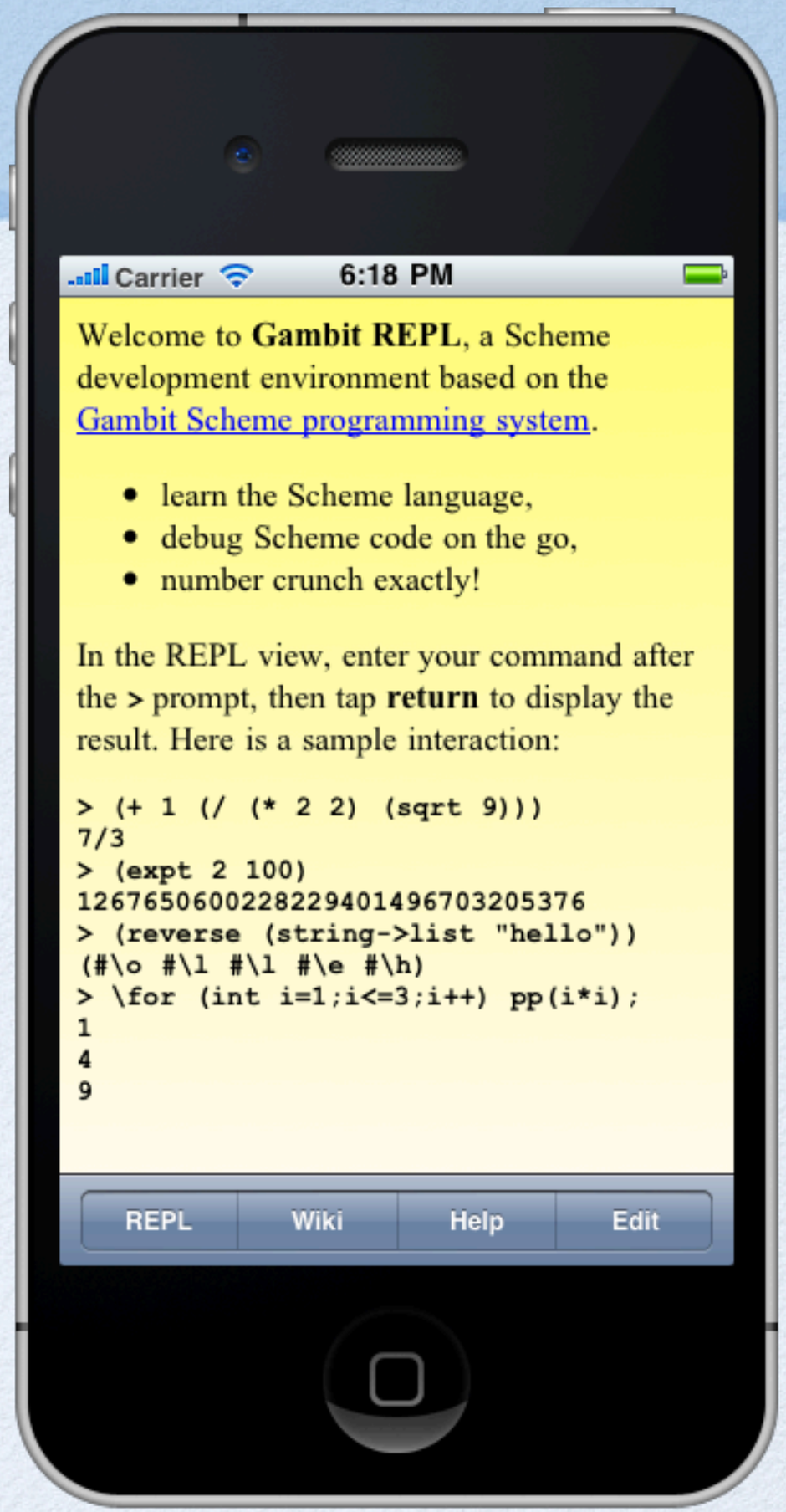
Overview

- User interface
- Development
- Implementation

User Interface

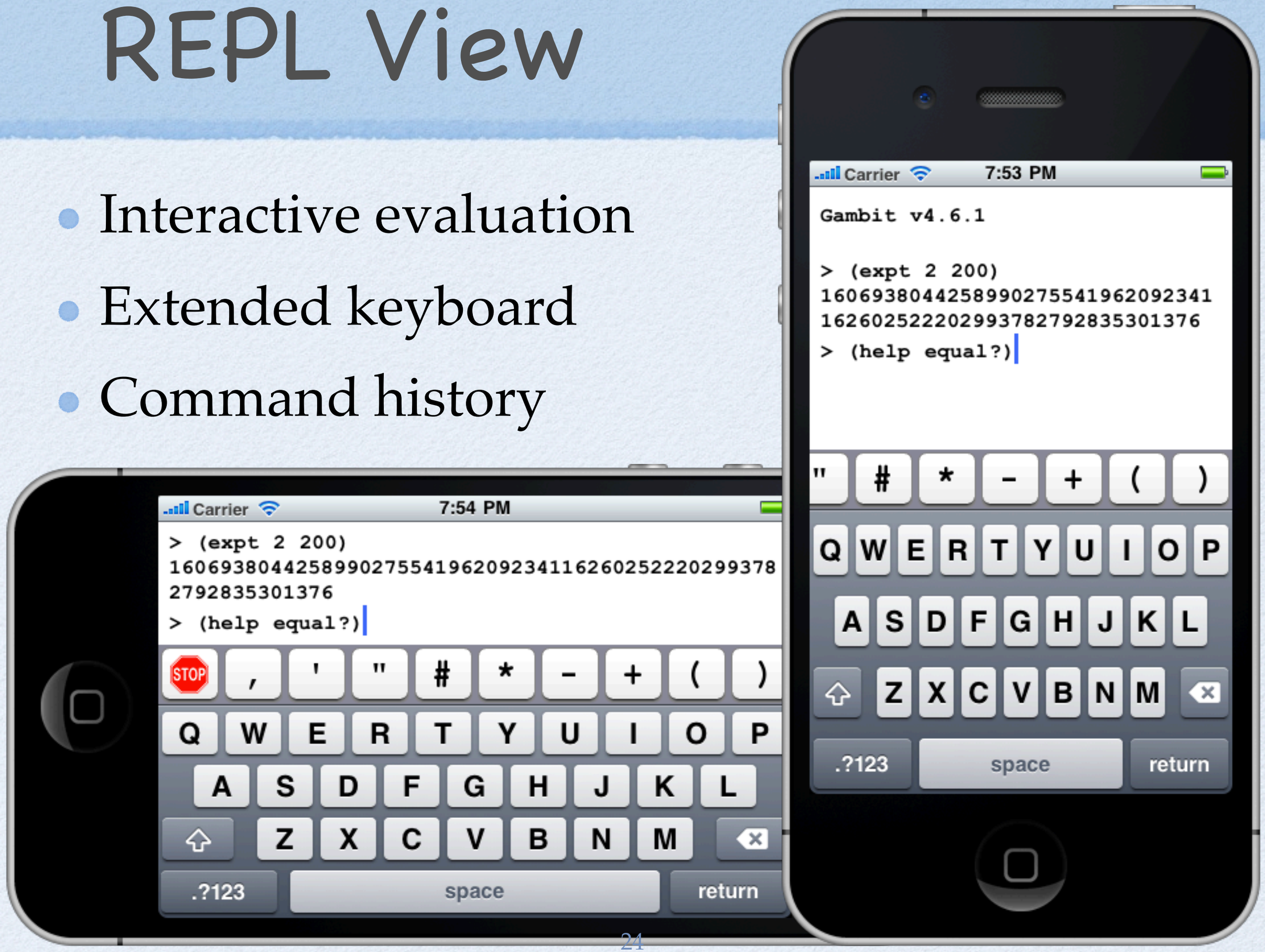
Splash Screen

- Four views:
  - REPL
  - Wiki
  - Help
  - Edit



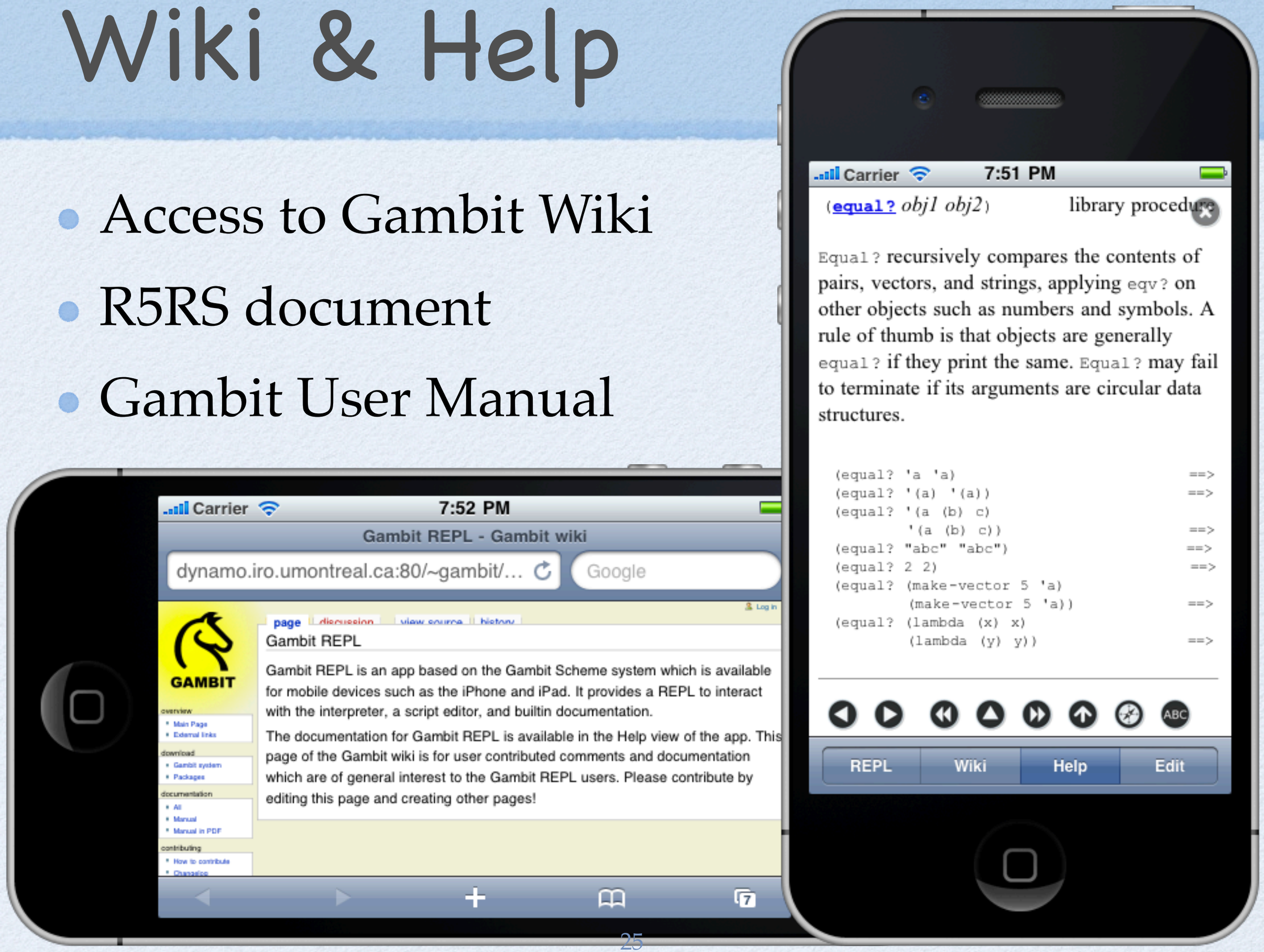
REPL View

- Interactive evaluation
- Extended keyboard
- Command history



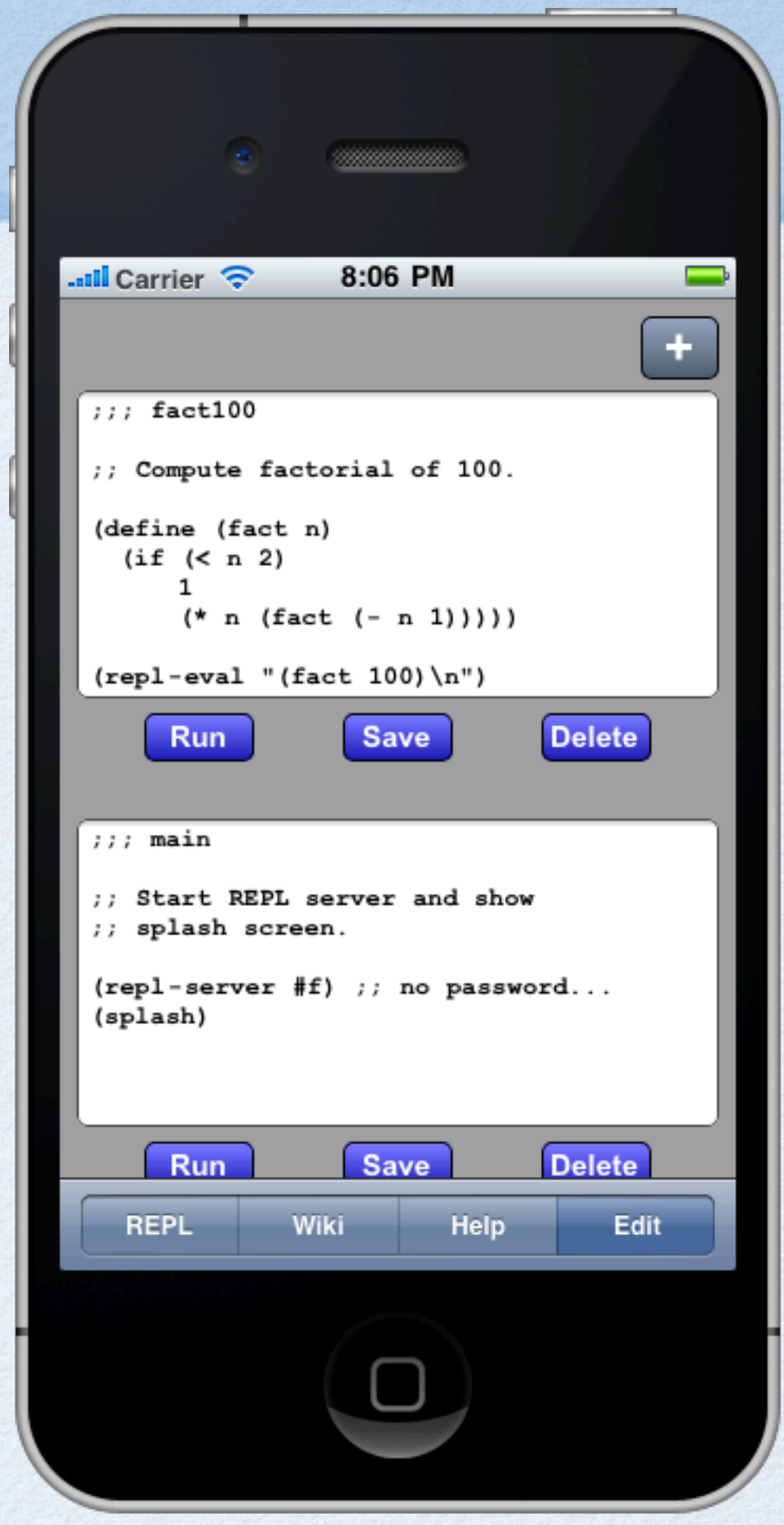
Wiki & Help

- Access to Gambit Wiki
- R5RS document
- Gambit User Manual



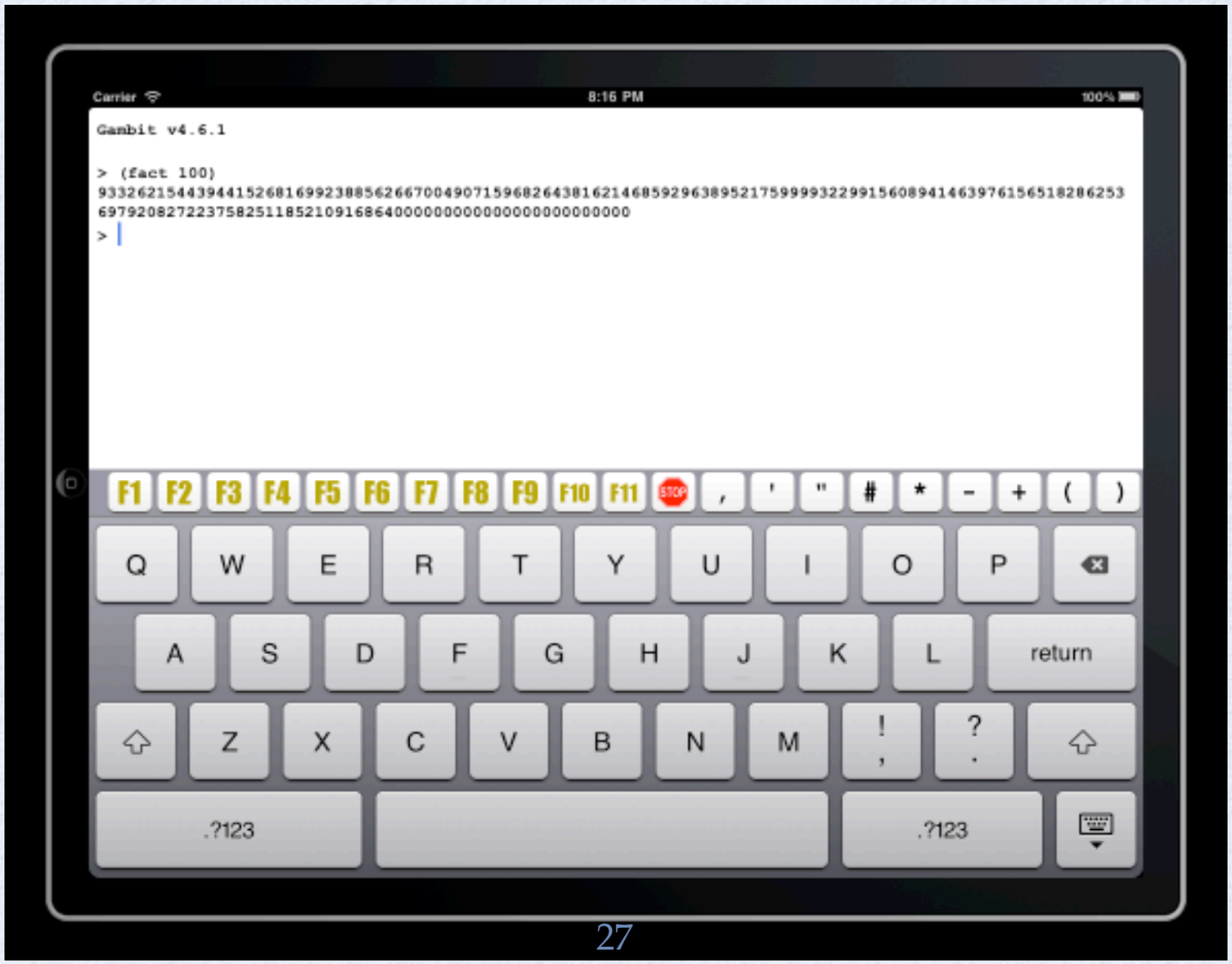
Edit View

- Script editor
- Add/Run/Save/Delete
- Each script has a name
  - "main" run at startup
  - "F1" run on F1 key, ...
- Save goes to "Documents"



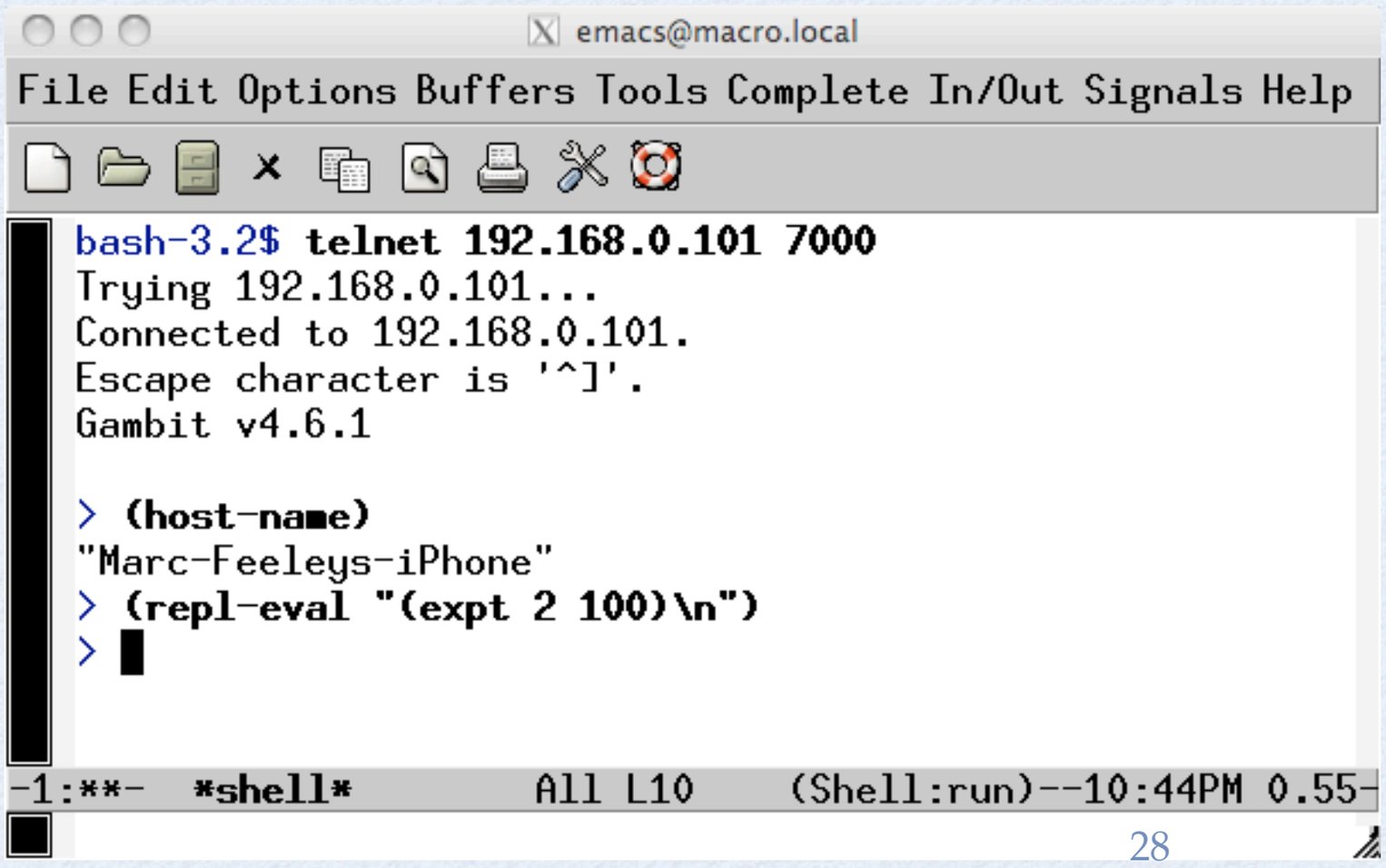
iPad Version

- Larger area is good enough for useful work
- Programmable function keys (F1 .. F12)



REPL Server

- Allows remote debugging
- Telnet to port 7000
- Multiple concurrent REPLs



Implementation

Highlights

- **gsc** + XCode 4
- Code is Scheme (5 KLOC) and ObjC (1 KLOC)
- Main app is in ObjC which calls into compiled Scheme code for processing UI "events"
- Most views are `webViews`
  - Dynamic generation of content as HTML
  - Intercept user "events" using `shouldStartLoadWithRequest`
- REPL view is a `textView`

Hardest Parts

- Supporting Gambit's green threads
  - Main app periodically calls Scheme `heartbeat` function to let the Scheme thread scheduler execute some threads
    - The `heartbeat` function returns the amount of time before the main app needs to call it again (it depends on the presence of runnable threads, the next sleep timeout, etc)
  - Implementation of wiki API to access the script repository (dealing with response parsing, timeouts, etc)

Related Apps

- iOS:
  - *Pixie Scheme* (iPad only, interpreter in C++)
  - *iScheme* (buggy, interpreter in JavaScript)
  - Apps for other languages: Lua (*Codify*, *iLuaBox*), Python (*Python Math*), Basic (*iBasic*, *HotPaw Basic*), JavaScript (*ExecScript*, *JSInt*)
- Android:
  - *Gambit for Android* (port of Gambit REPL v1)
  - *Scheme Droid*, *Clojure REPL*, *Ruboto IRB*, ...