

Building Wraith Scheme From Scratch



Jay_Reynolds_Freeman@mac.com
http://web.mac.com/Jay_Reynolds_Freeman
(especially the “Software” page)

This isn't a talk ...

It's a sideshow:

SEE – ALIVE – what kind of
FOOL would build an ENTIRE
R5 Scheme from a CLEAN
SHEET OF PAPER!!!

Note to self:
Take a bow.

Wraith Scheme is:

- ◆ An “R5” Scheme, with enhancements, for the Apple Macintosh.
- ◆ Executables: 32-bit/Tiger-or-better and 64-bit/Snow-Leopard-only.
- ◆ The 64-bit version is open source.
- ◆ See my web site’s “Software” page.

Pixie Scheme III is:

- ◆ Wraith Scheme stripped down for the Apple iPad. (No parallel processing and no file-system access.)
- ◆ One of the first programming tools for non-jail-broken iPads.
- ◆ Also open source – see my web site's “Software” page.

Pixie Scheme was:

- ◆ What started it all.
- ◆ An R3 Scheme for the early Macintosh.
- ◆ Written starting in 1987, first release 1988.
- ◆ Still available, as a courtesy to collectors of very old Macintoshes.
- ◆ See my web site, et cetera.

Incidentally...

“Wraith” was a cat.

“Pixie” was also a cat.

I like cats.

Can you tell?

Who am I?

- ◆ Used astrophysicist. (We are cheap!)
- ◆ Career programmer.
- ◆ Modest experience in Lisp/AI.
- ◆ I am pretty much informal: Feel free to scream, throw things, and ask lots of embarrassing questions.

Sources for this talk:

- ◆ Wraith Scheme has nearly 300 000 words of documentation.
- ◆ Half way between “Pride and Prejudice” and “War and Peace”.
- ◆ Links on my web site’s Software page.
- ◆ Those files are also in the application, and in the open-source release.

Slides for this talk:

- ◆ http://public.me.com/jay_reynolds_freeman
- ◆ The file is “WraithScheme.2.key”, for Apple’s “Keynote” application.

Features of interest:

- ◆ Some architectural notions of how to put a Lisp system together.
- ◆ How I designed it.
- ◆ How I built it – got it up and running.
- ◆ What has happened to it since then, notably parallel processing.

Some History – In 1987:

- ◆ 1 MByte was a LOT of memory.
- ◆ 10 MHz was a fast processor.
- ◆ A gnu was a silly-looking African animal.
- ◆ “Open Source” was a failure mode in MOSFET transistors.

I wanted a Lisp:

- ◆ To learn more about the language.
- ◆ To enhance for personal projects.
- ◆ I picked Scheme because it was compact, elegant and powerful.
- ◆ I wrote my own because I couldn't find usable source for an implementation in the microcomputer world.

Development Environment:

- ◆ I had a shiny new 1987 Mac II with Apple's Macintosh Programmers' Workshop.
- ◆ 16 MHz Motorola 68020. (Yes, MHz.)
- ◆ 5 MByte RAM. (Yes, MByte.)
- ◆ 80 MByte hard disk. (Yes, MByte.)
- ◆ Stripped-down Unix-like tool set.

Early Classic Mac OS:

- ◆ A decent early microcomputer version of the classic Xerox PARC GUI.
- ◆ Not Unix!!
- ◆ But also, not Windows!!
- ◆ And certainly not MS-DOS!!

Pre-ANSI C:

- ◆ Not C++, just plain C.
- ◆ I didn't like Apple's object-oriented Pascal, or think it was going anywhere.
- ◆ Straight C supports object-oriented programming very well.
- ◆ You hand-write the kind of code a C++ preprocessor would generate.

Planning Was The Key:

- ◆ I was in the middle of moving my residence when I started Pixie Scheme.
- ◆ My PC was in storage.
- ◆ The move kept getting delayed.
- ◆ For a few months, all I could do was do homework and design on paper.
- ◆ The enforced delay made me think.

Planning Was The Key (II):

- ◆ I am NOT saying, “Get it all right before you start writing code” – the agile folks are right, the “Waterfall” design method does not work and never did work.
- ◆ I do say, “Do enough homework to understand the choices open to you.”
- ◆ I also say, “Look before you leap.”

Two Kinds of Planning:

- ◆ Static Design: What will the finished system look like?
- ◆ Dynamic Planning (my turn of phrase): What is an efficient and relatively risk-free path for bringing it up?
- ◆ They are related: Choose static design in part for ease of bring-up.

Static Design: (what it will look like)

What Kind of System:

- ◆ Interpreter or compile-to-something?
 - ◆ Compile-to-... was in its infancy then.
 - ◆ Compiled systems may not port well.
What if I had a world class compiled Scheme for the Motorola 68000 ... ?
- ◆ Interpreters help with rapid program development of Scheme programs.

What Kind of System (II):

- ◆ I wanted compilation in the interpreter.
- ◆ Input and output are S-expressions.
- ◆ The compiler is a Scheme program that runs in the interpreter.
- ◆ Does optimizations – macros, bindings.
- ◆ Most useful if you can promise that a binding won't change before run-time.

What is a Scheme Object:

- ◆ You need to know what an object is:
 - ◆ For type-checking and avoiding errors.
 - ◆ To display it, et cetera.
 - ◆ To go past it in Scheme main memory (if objects are of different size).
 - ◆ To find pointers to dereference.

What is a Scheme Object (II):

- ◆ Many possibilities:
 - ◆ Big bag of pages – identify by address region.
 - ◆ Tagged object, with tag that identifies it.
- ◆ I chose a tagged-object system.
- ◆ “Tag” typedef and field values are vital.

Tagged avals (my term):

- ◆ You have a one-word something and some bits that say what it is. (Atom? Pointer? What kind? Cdr-coded?)
- ◆ The word can be either data – an rval (right value) – or a pointer – an lval.
- ◆ Hence “aval” – ambidextrous value.

Lots of bits:

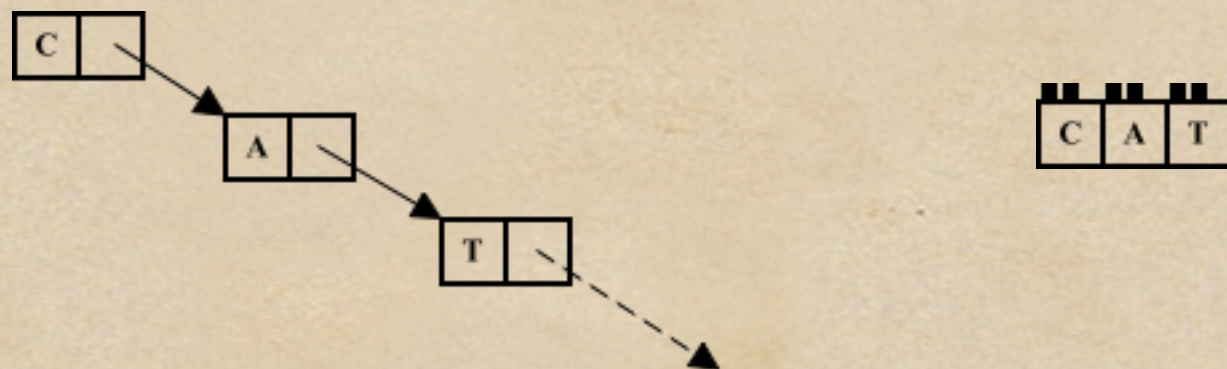
- ◆ Historically, tag bits have been a scarce resource and used parsimoniously.
- ◆ C structs use word alignment, so a tagged value gets a word's worth of tag bits whether you like it or not.
- ◆ So make lemonade – I have found lots of useful things to do with all those bits.

RAM Was Scarce:

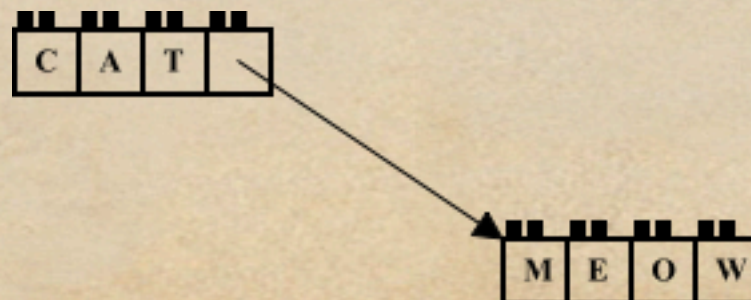
- ◆ You have all those tag bits, so cdr-code.
 - ◆ Use two bits and a type of object called a “forwarding pointer”, e.g.:
 - ◆ 00: Object is not in a list.
 - ◆ 01: Object is in list, cdr is nil.
 - ◆ 10: Object is in list, cdr is next.
 - ◆ 11: This bit combination unused.

RAM Was Scarce (II):

- ◆ The system automatically dereferences forwarding pointers.
- ◆ Two ways to make a list out of a cat:



- ◆ This list has two cdr-coded sections:



RAM Was Scarce (III):

- ◆ No swapping (except overlays).
- ◆ The garbage collector did not need to worry about swapping, since it couldn't.
- ◆ Now, with lots of memory, in-memory Garbage-collecting is fine. Yippee!
- ◆ The garbage collector cdr-codes as it works. That was a pain to get right.

Interpreter Architecture:

- ◆ Three parts: Model, View, Controller.
- ◆ Apple didn't call them that: View and Controller sort of merged into one.
- ◆ The Model was the part that ran Scheme – the read/eval/print loop.
- ◆ I call the part that runs Scheme the “Scheme Machine”.

Interpreter Architecture (II):

- ◆ Separate GUI facilitated bring-up and porting. The same Scheme Machine has run with six different GUIs: Shell in MPW, Classic MacOS, Unix shell, Unix with nCurses, Mac OS X, and iOS.
- ◆ <boast> Having a clean API between the Scheme Machine and the GUI was a very good design decision. </boast>

The Scheme Machine:

- ◆ My background influenced my choices for Scheme Machine architecture.
 - ◆ Moderate hardware experience.
 - ◆ I had done a lot with microcode.
 - ◆ What's microcode? Once popular ...

The Scheme Machine (II):

- ◆ Microcode is an architecture / language layer between assembler and hardware.
- ◆ Usually, microcode I/O bit corresponds to Vcc/ground on a specific wire.
- ◆ In some machines (Xerox PARC), microcode could call out to high-level language routines when necessary.

The Scheme Machine (III):

- ◆ "Functional Programming Application and Implementation" (Peter Henderson, Prentice-Hall, 1980) described:
- ◆ An applicative subset of Scheme.
- ◆ Running on a virtual Scheme machine described at the Register Transfer Level (RTL), with an assembly language.

SECD Machine:

- ◆ Henderson used but did not invent.
- ◆ Four registers point into Scheme heap.
 - ◆ S: Stack (evaluation stack, not call stack) – stack top is accumulator.
 - ◆ E: Environment – points to the current environment(s), where you look up bindings.

SECD Machine (II):

- ◆ C: Continuation – instructions left in the current basic block.
- ◆ D: Dump – like a call stack – each thing on it is a saved (S, E, C) triple.
- ◆ I added more registers, and during bring-up not all of S, E, C, and D existed, or did not point into the heap.

The Scheme Machine (IV):

- ◆ I planned to implement Henderson's virtual machine using C preprocessor macros for the assembly-language instructions.
- ◆ The macros would expand into C, which would be the microassembly language for the virtual machine.

The Scheme Machine (V):

- ◆ The “microassembler” could call other C routines of arbitrary complexity.
- ◆ Scheme primitive operations would be implemented as virtual-machine “assembly language” instructions.
- ◆ This layering simplifies the problem and facilitates changing the implementation.

An Example:

- ◆ In Scheme, you might ask the interpreter to evaluate

$(+ 2 2)$

- ◆ The evaluator calls up some code that implements “+”.
- ◆ What does that code look like?

An Example – C with macros:

```

/**** addCode -- Add the numbers stacked and return the sum,
                        except that if there are no numbers, return zero. ****/

PROC( addCode )

    ZERO_FIXNUM_P
    JUMP_FALSE( oneOrMore )
    CONTINUE

oneOrMore:
    ONE_FIXNUM_P
    JUMP_TRUE( noMore )
    EXCHANGE( R, STACK( 1 ) )
    ADD
    SWAP
    DECREMENT_R
    JUMP_LABEL( oneOrMore )

noMore:
    POP
    CONTINUE

END_PROC
```


An Example – preprocessed:

```
void addCode() {  
  
    B = ( (LongWord)R.u.c == 0 );  
    if( ! B ) goto oneOrMore;  
    return;  
  
oneOrMore:  
    B = ( (LongWord)R.u.c == 1 );  
    if( B ) goto noMore;  
    exchangeFunction( &(R), &((S[1])) );  
    add();  
    Swap();  
    (R.u.c = R.u.c - 1);  
    goto oneOrMore;  
  
noMore:  
    (R = *S++);  
    return;  
  
}
```


An Example – side by side:

```
/**** addCode -- Add the numbers stacked and return the sum,  
except that if there are no numbers, return zero. *****/
```

```
PROC( addCode )
```

```
    ZERO_FIXNUM_P  
    JUMP_FALSE( oneOrMore )  
    CONTINUE
```

```
oneOrMore:
```

```
    ONE_FIXNUM_P  
    JUMP_TRUE( noMore )  
    EXCHANGE( R, STACK( 1 ) )  
    ADD  
    SWAP  
    DECREMENT_R  
    JUMP_LABEL( oneOrMore )
```

```
noMore:
```

```
    POP  
    CONTINUE
```

```
END_PROC
```

```
void addCode() {
```

```
    B = ( (LongWord)R.u.c == 0 );  
    if( ! B ) goto oneOrMore;  
    return;
```

```
oneOrMore:
```

```
    B = ( (LongWord)R.u.c == 1 );  
    if( B ) goto noMore;  
    exchangeFunction( &(R), &(S[1])) );  
    add();  
    Swap();  
    (R.u.c = R.u.c - 1);  
    goto oneOrMore;
```

```
noMore:
```

```
    (R = *S++);  
    return;
```

```
}
```


An Example – comments:

```
void addCode() {                                     // COMMENTS:

    B = ( (LongWord)R.u.c == 0 );                   // B and R are virtual
    if( ! B ) goto oneOrMore;                       // machine registers ...
    return;

oneOrMore:
    B = ( (LongWord)R.u.c == 1 );
    if( B ) goto noMore;
    exchangeFunction( &(R), &((S[1])) );           // “exchangeFunction” and “swap” are inlines
    add();                                           // “add” does LOTS of work; it must cope
    Swap();                                         // with 64-bit integers, IEEE 64-bit floats,
    (R.u.c = R.u.c - 1);                           // and with complexes and rationals. “add”
    goto oneOrMore;                                // is (now) written in C++ ...

noMore:
    (R = *S++);                                     // stack operation as a macro
    return;

}
```


An Example – part of add:

```
void add()
{
    Boolean bothExactP;
    Boolean sameSignP = false;
    LongWord scratch = 0;
    LongWord newNumerator = 0;
    LongWord commonDenominator = 0;
    LongWord firstTermNumerator = 0;
    LongWord secondTermNumerator = 0;
    Boolean overflowed = FALSE;

    switch( coerceTwoCanonically( FALSE, &bothExactP ) ) {
        case ALL_LONG_RATNUMS:
            lowestTerms();
            Swap();
            lowestTerms();
            Swap();
            SPushR();
            SPushR();
            S[0] = S[2];
            numeratorAndDenominator(); // R: d2; S: n2; r1; r2; r1
            SPushR();
            R = S[2]; // R: r1; S: d2, n2; r1; r2; r1
            numeratorAndDenominator(); // R: d1; S: n1, d2, n2; r1; r2; r1
            commonDenominator = lcm1NoFail( R.u.c, (S[1]).u.c, &overflowed );
    }
```


An Example – thoughts:

- ◆ The C/C++ code that the preprocessor macros expand into, and the C/C++ code for the big “add” function, have changed many times.
- ◆ The virtual machine assembly language for “addCode” has not changed in twenty years.

An Example – thoughts (II):

- ◆ If we were speaking of real machines, we would be saying that I had introduced many new processors (virtual ones) with the same virtual processor architecture at the assembly-language level.
- ◆ That is a win for simplicity and layering.

An Example – thoughts (III):

- ◆ From top down, the layers are:
 - ◆ Scheme source code (like “(+ 2 2)”)
 - ◆ Scheme primitives (+, list, cons ...)
 - ◆ Assembler (PUSH, ADD, POP ...)
 - ◆ Microcode (C/C++ routines)
- ◆ Layering separates one big complicated mess into smaller, simpler messes. Win!!

The Scheme Machine (VI):

- ◆ Other pieces:
 - ◆ Allocator: Manages main memory.
 - ◆ Garbage collectors: Both the stop-and-do-it-all variety and (,in Wraith Scheme) the generational variety.
 - ◆ Evaluator: A big fussy loop.

The Scheme Machine (VII):

- ◆ More pieces:
 - ◆ Compiler: To threaded code, with as much symbol evaluation done in advance as possible.
 - ◆ ObList: There is only one instance of any given symbol, so symbol equality is a matter of pointer equality.

Dynamic Design: (bringing up baby)

Dynamic Design Objectives:

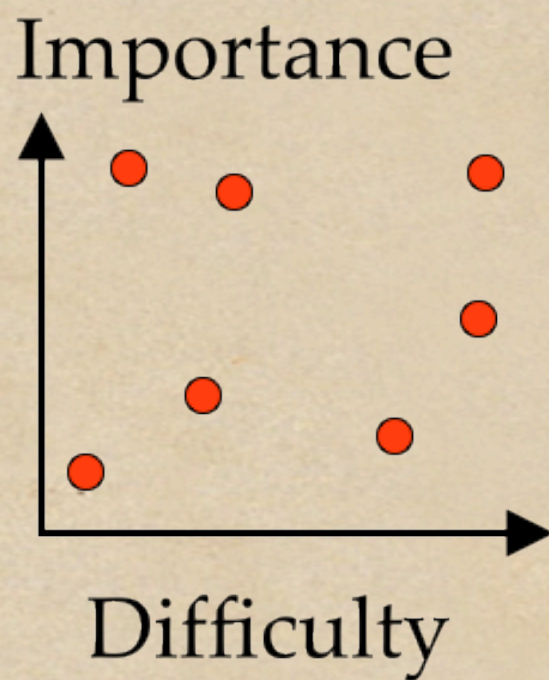
- ◆ Be able to build the pieces one at a time
– no messy looping dependencies.
- ◆ Be able to test the pieces one at a time.
- ◆ Bring up code so as to always have something to test, and test it!
- ◆ Build a test suite as you go along.
- ◆ Keep clean interfaces between pieces.

Dynamic Design Approach:

- ◆ This all sounds very familiar ...
 - ◆ Object-oriented design.
 - ◆ Agile software development.
 - ◆ Et cetera ...
- ◆ But this was 1987.
- ◆ Many of us had learned to do this stuff without benefit of tools or formalisms.

Top Left Corner Analysis:

- ◆ Plot tasks by difficulty and importance:



- ◆ Start in top left corner.
- ◆ On good days, move right.
- ◆ On bad days, move down.
- ◆ On terrible days, sleep ...
- ◆ Tasks move around as you progress.
- ◆ Important: Needed next, or wanted next.

Bring Up Order:

- ◆ Start in a stdio shell. (MPW, not Unix.)
- ◆ Define an incomplete RTL machine.
- ◆ Test simple Scheme primitives in separate C programs, using macros.
- ◆ Add a memory allocator – can test till it fills up, even with no garbage collection.
- ◆ Define classes for primitives and objects.

Bring Up Order (II):

- ◆ Create a startup world with primitive bindings. Need world save / world load.
- ◆ Write a reader. No lex/yacc for micros!
- ◆ Write an evaluator. A mess.
- ◆ Write garbage collector.
- ◆ Et cetera.
- ◆ Add GUI at “leisure”.

Incomplete RTL Machine:

- ◆ I used an accumulator (register R) as the top of the evaluation stack.
- ◆ I added some more registers, too.
- ◆ For starters, with no heap, made the evaluation stack a plain memory array.
- ◆ Added “assembly language” instructions to load / display registers.

Incomplete RTL Machine (II):

- ◆ An individual C program might:
 - ◆ #include register definitions and macro files.
 - ◆ in main(), load registers, #include file "PROC(add)", and display results.
 - ◆ Thus one can get Scheme primitives going, in terms of "microassembler".

Storage Allocator:

- ◆ Simple storage allocator is an array with a pointer dividing used from unused.
- ◆ Needs assembler instructions to get at it, e.g., “CONS”, “NILCONS”, etc.
- ◆ RTL machine got an address register.
- ◆ With no garbage collection you just test till memory fills up. Big tests possible.

Scheme Primitives Class:

- ◆ I defined in essence a two-layer class system for Scheme primitives (“+”, etc.).
- ◆ Each instance contains:
 - ◆ Pointer to code to do the work.
 - ◆ Index of primitive name in a table.
 - ◆ A couple slots about what it returns.

...

Scheme Primitives Class (II):

- ◆ (Variant/subclass) some numbers to tell how many arguments it can have.
- ◆ A pointer to an array of functions to type-check the arguments.
- ◆ A pointer to an array of text strings for error messages. (They describe the kind of arguments expected.)

Scheme Objects Class:

- ◆ Each object type (integer, pointer to pair, et cetera) is an instance of a class whose instance variables include:
 - ◆ A cursor into a list of print names.
 - ◆ A value for the tag field that identifies object types.
 - ◆ A length for the object, except ...

Scheme Objects Class (II):

- ◆ ... if the given length is zero, the next slot is a pointer to a function that can find the length (e.g. by looking at the character count of a string).
- ◆ A pointer to a “display” function.
- ◆ A pointer to a “write” function.
- ◆ A pointer to an “inspect” function.

Scheme Objects Class (III):

- ◆ A pointer to a function that flips a big-endian instance of the object to little-endian.
- ◆ A pointer to a function that flips a little-endian instance of the object to big-endian.
- ◆ These last make worlds cross-platform.

Worlds:

- ◆ When Wraith Scheme starts up, the initialization routines will load all the primitive functions into an empty Scheme main memory.
- ◆ Then, if a world file (Scheme main memory image) is provided, Wraith Scheme will load that.
- ◆ What if not?

Worlds (II):

- ◆ Constructing a complete world is a matter of loading a carefully crafted series of Scheme files into the bare-bones primitives-only world, to build up the rest, and then saving the result.
- ◆ The saved world so constructed then gets bundled with the rest of the application – the version that ships.

Worlds (III):

- ◆ Gotcha #1: Wraith Scheme can't open a GUI with the bare-bones world: So build the complete world with the terminal-shell version of Wraith Scheme.
- ◆ Gotcha #2: Wraith Scheme bootstraps now – you need a world to build one: Fine, but losing all worlds means you must create chicken or egg, brand new.

Reader:

- ◆ Yes, I did indeed hand write a Scheme reader (lexer and parser).
- ◆ No, I would not have hand written it if I had had a version of lex/yacc that ran on my Mac II.
- ◆ The only reason I might recommend that you hand write one is that misery loves company.

Evaluator – The E in REPL:

- ◆ The evaluator is fussy to get right.
- ◆ It is a push-down automaton that loops.
- ◆ It cannot recurse: It is written in C.
- ◆ It uses a stack of automaton states, and
- ◆ I have merged the SECD S and D registers of the SECD – like a C stack.
- ◆ Tail-call optimization reuses frames.

Garbage Collection:

- ◆ The strategy here was:
 - ◆ Write a simple one.
 - ◆ Make it the default.
 - ◆ Write other, hopefully better ones.
 - ◆ Make them optional at first.
 - ◆ Make them defaults after testing.

Garbage Collection (II):

- ◆ First: Dual-space, stop and copy, no cdr-coding during garbage collection. The first collector is now long gone.
- ◆ Second: Ditto, with cdr-coding.
- ◆ Third: Generational GC with two generations plus “tenured”; do stop-and-copy when necessary.

Pixie Scheme summary:

- ◆ Interesting experience creating it.
- ◆ Got a little respect as shareware.
- ◆ 1980s platforms were inadequate for the projects I wanted to do.
- ◆ So I shelved my Scheme projects for about 15 years.

Fast-forward to 2006:

- ◆ 15 years had replaced “mega” by “giga” everywhere it counted.
- ◆ Replaced the MacOS 6 GUI in Pixie Scheme with stdin/stdout shell I/O.
- ◆ Ported to C++ (most of it compiled!)
- ◆ Added a modern Mac/Cocoa GUI.

Wraith Scheme was born:



... and moved forward.

Early developments:

- ◆ R3, R4 and R5.
- ◆ Added assorted small features.
- ◆ Wraith Scheme 1.30 – still 32-bit – was a big jump, to parallel processing.

Parallel Processing:

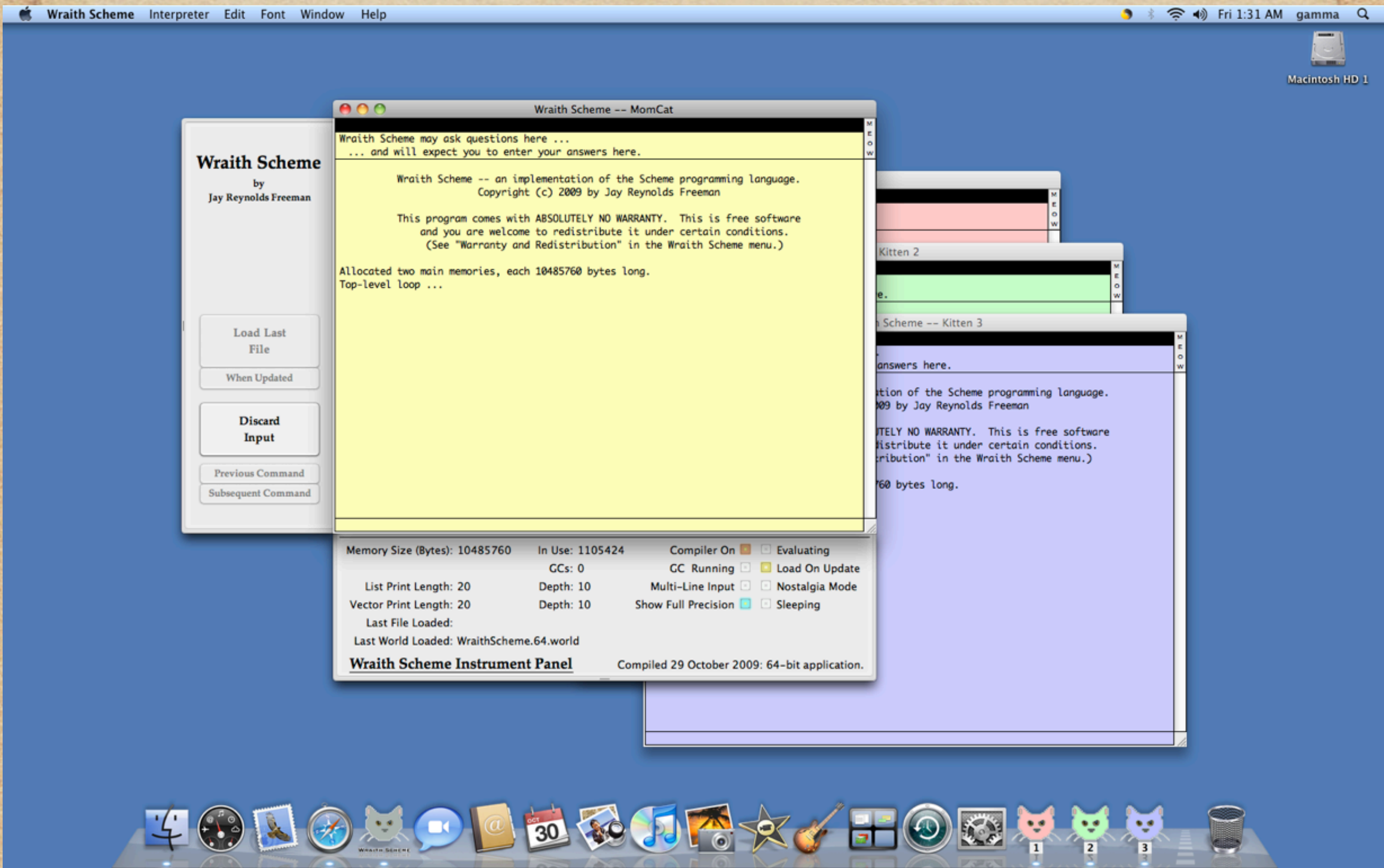
- ◆ Separate UNIX processes.
- ◆ Each has its own Mac GUI.
- ◆ Shared Scheme memory, locked on a word-by-word basis.
- ◆ Interprocess communication by read queue as well.
- ◆ A privileged process for some serial tasks.

Separate UNIX processes:

- ◆ Threads share data by default.
- ◆ Processes separate data by default.
- ◆ Processes leverage the UNIX scheduler.
- ◆ If one Scheme process has a problem, you can use others to investigate.

Each has its own GUI:

- ◆ One main window per process.
- ◆ One menu bar per process.
- ◆ One read/eval/print loop per process.
- ◆ I rarely need a GUI for each process ...
- ◆ ... but it is sometimes handy.
- ◆ I use color to distinguish GUIs.



Shared main memory:

- ◆ Use `.zerofill` / `-segaddr` to reserve lots of address space before the system storage allocator gets started.
- ◆ Use “`mmap`” to create large shared heaps, with a file for swapping out.
- ◆ Documentation for “`mmap`” is perhaps inadequate for beginners.

Main memory locking:

- ◆ Lock to avoid messing up the heap by simultaneous low-level operations.
- ◆ Locks are held for short time.
- ◆ No code holding a lock ever asks for an additional one (to avoid deadlocks).

Locking continued:

- ◆ Obtain a lock by looping on `OSAtomicSwapAndCompareLong`.
- ◆ Lock sets an 8-bit tag field to show where in the source the lock was made.
- ◆ Another field identifies the locking process.
- ◆ That information helps with debugging.

Read queues:

- ◆ Every process has a queue of S-expressions – already parsed – into which any process may add items.
- ◆ The “read” part of the read/eval/print loop takes S-expressions from the queue before looking at the keyboard.

A process for serial tasks:

- ◆ I use a feline metaphor to name Wraith Scheme processes.
- ◆ A privileged process – the “MomCat” – supervises certain serial system tasks, such as garbage collection, world saves and world loads.
- ◆ The other processes are “kittens”.

More recently added:

- ◆ 64-bit implementation.
- ◆ Generational garbage collector.
- ◆ Simple class mechanism.
- ◆ Coarse-grained foreign-function interface, based on shared memory and interrupts.

Summary:

- ◆ Don't try this at home, kids!
- ◆ Decompose problems and layer the architecture.
- ◆ Use object-oriented design practice.
- ◆ Keep APIs crisp and clean.
- ◆ Build a test suite as you develop, and don't turn around without running it.

Summary (II):

- ◆ Wraith Scheme and Pixie Scheme III today are perfectly viable descendants of a program written in 1987.
- ◆ Software reuse/portability does work!
- ◆ About half of the Wraith Scheme (2011) source code is the same as for Pixie Scheme (1988).

Summary (III):

- ◆ That code has run in six GUIs: Shell in MPW, Classic MacOS, Unix shell, Unix with nCurses, Mac OS X, and iOS.
- ◆ It has run on six different processor architectures: Motorola 68000 and 68020, Power PC 32-bit, Intel 32-bit and 64-bit, and ARM 32-bit.

Summary (IV):

- ◆ Those architectures include both big-endian and little-endian.
- ◆ It has run in three different operating systems: Classic Mac OS, Mac OS X / Unix, and iOS / Unix.
- ◆ It has run in two different programming languages: Pre-ANSI C and C++.

Summary (V):

- ◆ I learned a tremendous amount while working on my Scheme systems.
- ◆ It was very satisfying to create them.
- ◆ And I had fun doing it!
- ◆ That probably makes me weird ...
- ◆ ... but I told you this was a sideshow.

Thanks for inviting us.

(Have a cookie.)



Jay_Reynolds_Freeman@mac.com

http://web.mac.com/Jay_Reynolds_Freeman