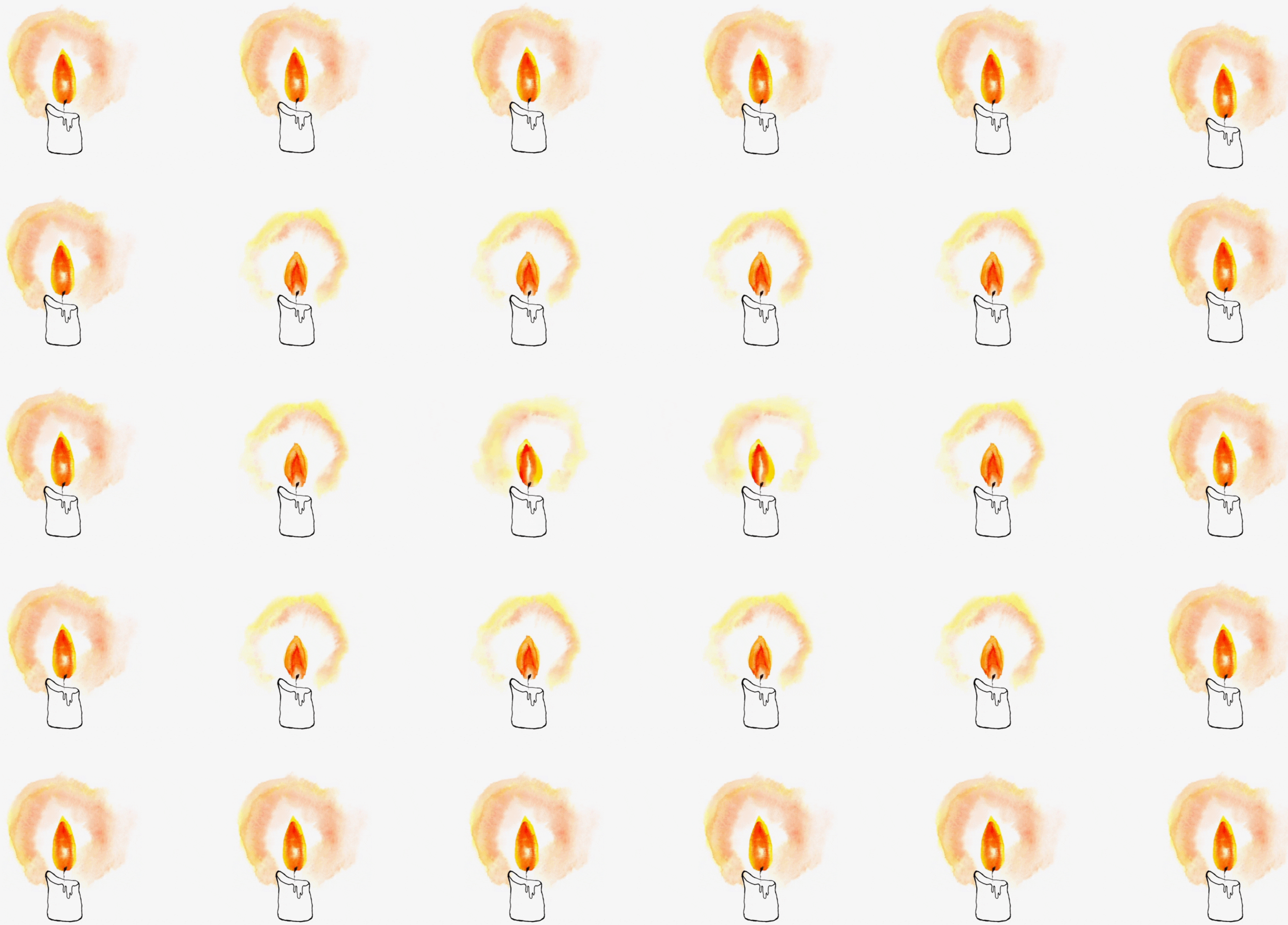
GAMBIT
@30

History

Marc Feeley

October 12, 2019

Université de Montréal



History/Overview/Metrics

Early History

- **1988**: the "SIS" (Straightforward Implementation of Scheme) is created as a term project for a Logic Programming course at Brandeis University
- **1989**: it is translated to Scheme and renamed
- "Gambit" becomes the basis for research on efficient implementation of Scheme on large shared-memory multiprocessors (see PhD thesis for details…)
- **1990**: first research paper on Gambit's VM (PVM)

SIS documentation

SIS Compiler documentation

(Straightforward Implementation of Scheme)

(version 0.1 for SUN, jan 26 1988)

By Marc Feeley

---

1. Using the compiler

The compiler comes with the following files:

```
  doc            this file
  sc             the compiler itself written in Quintus Prolog
  header.s       an assembly file containing the run-time system
  asm            command file to assemble the code generated by the compiler
  queens.scm     a couple of test files in scheme
  fib.scm
  tak.scm
  sort.scm
```

To use the compiler, write the scheme program you want to compile into a file; the file must have the extension '.scm' (for the sake of brevity let's call it 'source.scm'). Then start your Prolog and load 'sc'. Type the query 'ex.' to start the compiler. It will then ask for the name of the file to compile; type the file's name without the extension (ie. source). A trace of the compilation phases should appear. The file 'source.s' will be generated which contains an MC68000 assembly language program. Exit Prolog and type 'asm source' this will assemble the compiler's output and generate the file 'source', the executable image of the program. To execute, type 'source'.

It is possible to ask for the open-coding of certain procedure calls (presently only car, cdr, +, -, *, /, -1+). This can be done by typing 'integrate(all).' before typing 'ex.'. This speeds up execution by a small amount. However, open-coded procedures do not check the type of their arguments or result, so be careful. To remove open-coding type 'integrate(none).'. By typing the query 'debug(on).', the compiler will generate code that will enable 'write' to print the source of the procedures instead of just '#<procedure>'. Type 'debug(off).' to generate the usual code.

Although the compiler adheres fairly closely to R3RS there are certain restrictions. The most important of which are the lack of a GC and the small number of primitive procedures implemented. However, all special forms are implemented. You should read the comments in 'sc' for more information. Here is a list of the primitives which are implemented:

```
not, eq?, pair?, cons, append, length, car, cdr, set-car!, set-cdr!,
null?, =, <, >, +, -, *, /, -1+, force, write, newline, list, vector,
list->vector, memq, assq, symbol?, vector?, string?, procedure?, number?,
char?
```

In Quintus Prolog you can pre-compile 'sc' to save time on the following uses of the compiler. Simply type:

```
compile(sc).
```

when you are in Quintus (this gives some warning messages) and then type:

```
save('sc.bin').
```

On subsequent uses of the compiler, startup Quintus and then type the query:

```
restore('sc.bin').
```

The compiler itself is written in a fairly portable fashion. The main system dependent code is for I/O. Interface procedures have been written for I/O so only these have to be modified. The compiler uses 'retract' and 'assert' statements (to generate symbols and so forth), which means that some of the clauses have been declared 'dynamic'. On other Prologs the 'dynamic' declarations might have to be removed.

2. How does it work?

Compilation is a 3 step process:

```
  1 - Parse input (ie. generate parse tree for program)
  2 - Compile program (ie. generate intermediate code)
  3 - Generate assembly code (ie. transform interm. code to M68000 code)
```

2.1 Parsing

The first step consists in reading all the characters of the source file into a list. This is not very efficient but it simplifies parsing. The list is parsed using a DCG grammar description of Scheme. A parse tree of the program is generated by the parser. A simple list representation similar to Scheme's own representation for S-expressions is used. For example, the expression

```
    (DEFINE (weird x) (list x "ABC" 123 #t #(1 2 3)))
```

is represented by the Prolog structure:

```
    [define,[weird,x],[list,x,str([65,66,67]),123,boo(t),vec([1,2,3])]]
```

Note: uppercase characters are automatically transformed to lowercase.

2.2 Compiling to intermediate code

This step takes the parse tree of the program and generates the corresponding intermediate code (ie. pseudocode). Before the intermediate code can actually be generated, the expressions must first be put into a normal form. This simplifies case analysis in the code generator. The normalization of the expressions is done in 3 steps.

```
    1 - Macro expansion (to take care of derived special forms)
    2 - Alphatization (ie. renaming of the variables) and
        assignment conversion
    3 - closure analysis
```

Most of macro expansion is straightforward. It's simplicity stems from Prolog's ease of doing pattern matching and case analysis. The hardest case to handle is mutually or self-recursive expressions (such as 'letrec's, 'define's, etc.). I will explain this case using an example. Suppose the expression to compile is:

```
    (letrec ((a (lambda (x) (b (+ x (c 1)))))           ; def1
             (b (lambda (x) (if (d x 0) (a x) (c x))))  ; def2
             (c (lambda (x) (- x)))                     ; def3
             (d (lambda (x y) (< x y))))                ; def4
      (a (c -30)))                                      ; body
```

The compiler will start by constructing the variable dependency graph for the (local) variables in this expression. Thus for the example:

```
     a <-----> b                    ( x --> y  means x depends on y )
       \     / |
        \   /  |
         V V   V
          c    d
```

It then does a topological sort on the graph to find out in which order the variables should be bound. The topological sort also discovers the cycles in the dependencies; these must be handled in a special manner. Cyclical dependencies conversion involves a kind of 'Y' operator. For the example we would obtain:

```
    (let ((c (lambda (x) (- x))))
      (let ((d (lambda (x y) (< x y))))
        (let ((a (lambda (a b)
                   (lambda (x)
                     (let ((a (a a b))
                           (b (b a b)))
                       (b (+ x (c 1)))))))
              (b (lambda (a b)
                   (lambda (x)
                     (let ((a (a a b))
                           (b (b a b)))
                       (if (d x 0) (a x) (c x)))))))
          (let ((a (a a b))
                (b (b a b)))
            (a (c -30))))))
```

Think about it, it works... A more 'natural' conversion would be to generate an allocate/assign/use form (as explained in the R3RS) but our solution is entirely functional and through data-flow analysis the compiler could recover the original expression's semantics (presently the compiler is not that intelligent but it might come in the future).

After this phase, only the basic special form of expressions are left in the program, ie.

```
- constant reference       eg.  123
- variable reference       eg.  x
- assignement              eg.  (set! x 456)
- conditional expression   eg.  (if a b c)
- application              eg.  (list a b c)
- lambda-expression        eg.  (lambda (b) b)
```

Alphatization consists of renaming the local variables to prevent aliasing problems. In this step, assignments are also converted into a 'functional' equivalent by introducing cells which hold the values of mutable variables. This simplifies the handling of closures and continuations since after this conversion, the value of any (non-global) variable always stays the same after it has been bound. Only data structures are mutable.

The last normalization step is closure analysis. It consists in anotating each lambda-expression with the set of its closed variables (ie. the free, non-global variables which are referenced in its body). Space for these variables will be allocated in the closures generated for this lambda-expression.

Once the program is normalized, it is passed to a DCG grammar in order to generate the intermediate code. Code generation is case driven and is basically a post-order traversal of the normalized parse tree. However, several special cases are recognized for which efficient code can be generated. This includes code for:

- the application of a procedure bound to a global variable
- the body of lambda-expressions with no closed variables
- the application of a lambda-expression (ie. a 'let')
- a 'let' variable which is not used in the body
- ordering of arguments to evaluate trivial arguments last
  (ie. arguments which need only a small number of resources)

2.3 Assembly code generation

This step scans the intermediate code instruction stream and generates the appropriate MC68000 code for them. The intermediate code is machine independent where as the assembly code generated is MC68000 and 'run-time structure' dependent. In theory, only this part has to be modified to generate code for another machine.

In addition, this step is responsible for the 'dumping' of Scheme objects to the code file. These are most often the constants contained in the source program. For example, if the following expression were compiled:

```
(memq 'b '(a b c))
```

the symbols a, b and c and the list (a b c) would be 'dumped' to the code file (and the symbol b would be dumped only once).

3. Benchmarks

We have run the compiler on a few benchmark programs. No special declarations were used and the programs were run on a SUN3. Here are the results in secs:

tak	1.3
sort	1.7
queens (without the trace)	1.7
fib (10 times (fib 20))	4.1

It would be interesting to measure these results against other Scheme implementations.

Latest Gambit with C backend on a 2013 machine is 13000x faster...

4. Run-time structure

A quick look at the benchmarks shows that the code generated is fairly efficient. The system's efficiency is mainly due to a cleverly designed run-time architecture. It has been designed in a way that makes frequent operations perform quickly. Some of the most frequent operations in Scheme are:

- checking type of an object
- calling a procedure (very frequently the procedure comes from a global variable and has no closed variables)

In our system, objects are represented by tagged 32 bit pointers. The tag resides in the 3 least significant bits of the pointer. The coding is as follows:

```
    least 3 bits = 000   object is a 29 bit integer (ie. a fixnum)

    least 3 bits = 100   object is a pair whose address is given by the
                         pointer.  the address is where the car is and
                         the cdr is just BEFORE this address.

    least 2 bits =  10   object is a memory allocated object (other than
                         a pair) whose address is given by the pointer.
                         the word that precedes this address gives the
                         type of the object.

    least bit    =   1   object is either a special value ( (), #f, #t ),
                         a character or a 31 bit floating point number
                         the value 111..111 represents () and 111..101
                         represents #f.
```

With this coding integers can be added or substracted directly without first having to extract the type (multiplication and division require an additional 3 bit shift).

Pairs can also be manipulated easily. Indirect addressing can be used to access the car of the pair and predecrement addressing to access the cdr.

Procedures (which are memory allocated objects) can be called simply by jumping to the pointer. This is possible because all procedures (including true closures) start with machine code. True closures (ie. procedures with closed variables) are represented like this:

```
          <- 16bits ->
          +----------+----------+----------+----------+----------+----
    ---> |JSR opcode|          |          |          |          | ...
          +----------+----------+----------+----------+----------+----
                      \_________ _________/ \_________ _________/
                                |                     |
                      Pointer to closure's     First closed       ...
                              body           variable's value
```

Thus the caller does not have to distinguish simple procedures from true closures. Environment related processing is done automatically (and if needed) by the callee. This implies that there is no overhead when calling procedures with no closed variables (the most frequent case).

Testing for 'falsity' is also efficient. Remember, in Scheme both () and #f represent false. In some implementations they are the same object (which means you can't distinguish () from #f). In our system we wanted them to be different. The representation is such that () and #f are the 2 highest object values whose least significant bit is 1. By adding 3 and checking the carry flag one can determine if the object was () or #f (ie. the carry is on) or if it was something else (ie. the carry flag is off).

In order to have efficient type checks we have 3 dedicated data registers that always contain the following masks:

```
    d5 = 00000001000000010000000100000001 for checking fixnums
    d6 = 00010000000100000001000000010000 for checking pairs
    d7 = 01000100010001000100010001000100 for checking mem. alloc. obj.
```

To test if a given data register contains a given type, a single 'btst' (bit test) instruction is required. For example, branching to a label if d1 contains a pair can be performed with the following code:

```
        btst    d1,d6      (use last 5 bits of d1 to index d6)
        bne     label      (branch if bit tested was 1)
```

A very common type check is making sure that only procedure objects are called. With our representation this involves 7 MC68000 inctructions. It would be prohibitive to perform this check every time a procedure is called. In our system, a special trick is used for the frequent case of calling procedures bound to global variables. Global variables are allocated in a table (one of the address register points to this table). Each global variable occupies 6 bytes; 4 bytes for the variable's value and 2 bytes (just before the value) that encode the type of the value. These 2 bytes contain the opcode 'jmp' (long jump) if the value is a procedure and a 'trap' opcode if it isn't a procedure of if the type is not yet known. When a procedure bound to a global variable needs to be called a jump to the variable's opcode is done. Most of the time the value of the variable will be a procedure and it will be jumped to. If this is not the case the trap handler is called; it will determine if the value at the return address is a procedure or not. If it is, the opcode will be changed to 'jmp' and a jump to the procedure will be performed. If it is not, an error handler is called. The price one pays for this scheme is that assignments to global variables must also change the variable's opcode back to a 'trap' opcode. Since assignments are infrequent in Scheme programs this is not a high price to pay.

The procedure calling convention used in our system is as follows:

- the first 4 arguments are passed in registers (d1, d2, d3 and d4) and the rest in fixed memory locations.
- the register d0 is loaded with an argument count code just before jumping to the procedure.

The procedure called must check that it has been called with the correct number of arguments. To make this check efficient we have a special coding for the argument count:

number of arguments:	0	1	2	3	4	5	...
argument count code:	1	-1	0	4	5	6	...

When the argument count code is loaded into d0 the flags are automatically set in accordance to the code loaded. For 1 argument the N (negative) flag is set, for 2 arguments the Z (zero) flag is set. Thus a single branch-on-condition instruction is required for the check when 1 or 2 arguments are expected (these are frequent cases).

5. Future plans

In its present state, the compiler is mainly useful for learning about Scheme or Lisp compilation techniques. Real applications can not be developed due to the lack of a GC, debugger, support routines, etc. In addition, since the compiler is written in Prolog the system is far from interactive.

In the future (and if time permits), we plan on doing some of the following:

- rewriting the compiler in Scheme and cleaning up the code,
- adding a GC and a full bag of primitive procedures,
- turning the system into an interactive one (with read-eval-print loop),
- adding some 'intelligence' to the compiler.

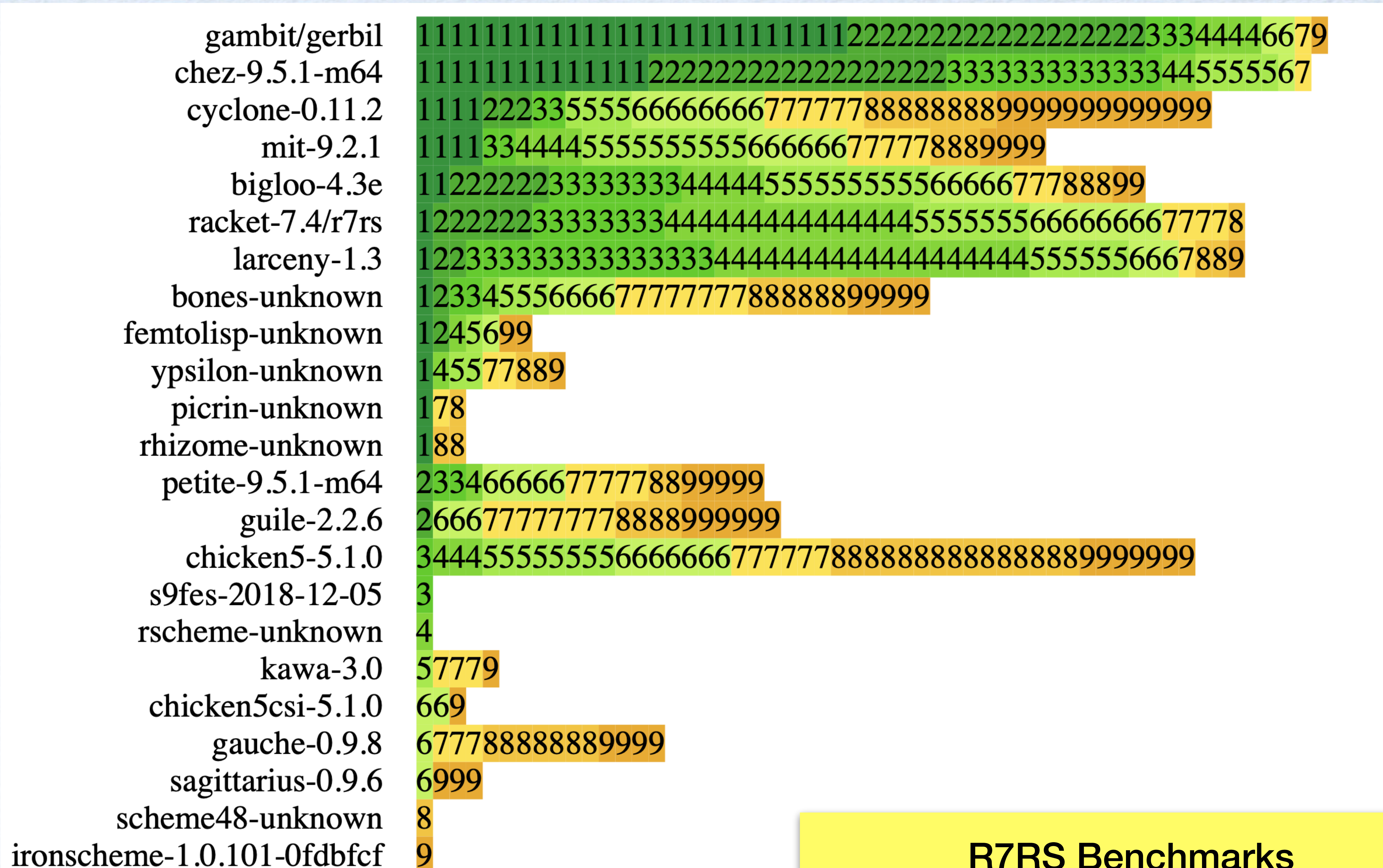
All comments regarding the SIS compiler are welcome (especially bug reports and suggestions). If you use the compiler or add some features to it please let me know...

Write to: feeley%brandeis.edu@csnet-relay

6. Acknowledgements

The compiler is the outgrowth of a course project for the 'Logic Programming' course taught by Tim Hickey at Brandeis University. We would like to thank Tim for his wonderful course and for suggesting this project.

Fast Forward 30 Years



R7RS Benchmarks
September 13, 2019

Fast Forward 30 Years

- **`gsi`**: interpreter optimized for debugging
- **`gsc`**: efficient compiler with several targets
  - C (most mature, is "reference implementation")
  - JS, Python, PHP, Ruby, go, Java
  - x86, ARM, RISC-V
- Gambit has been used widely for teaching, research, and in production (business, medical, CAD, games, building other programming language implementations e.g. JazzScheme and Gerbil)

Design Goals

- #1 - design a compiler/interp. I would like to use!

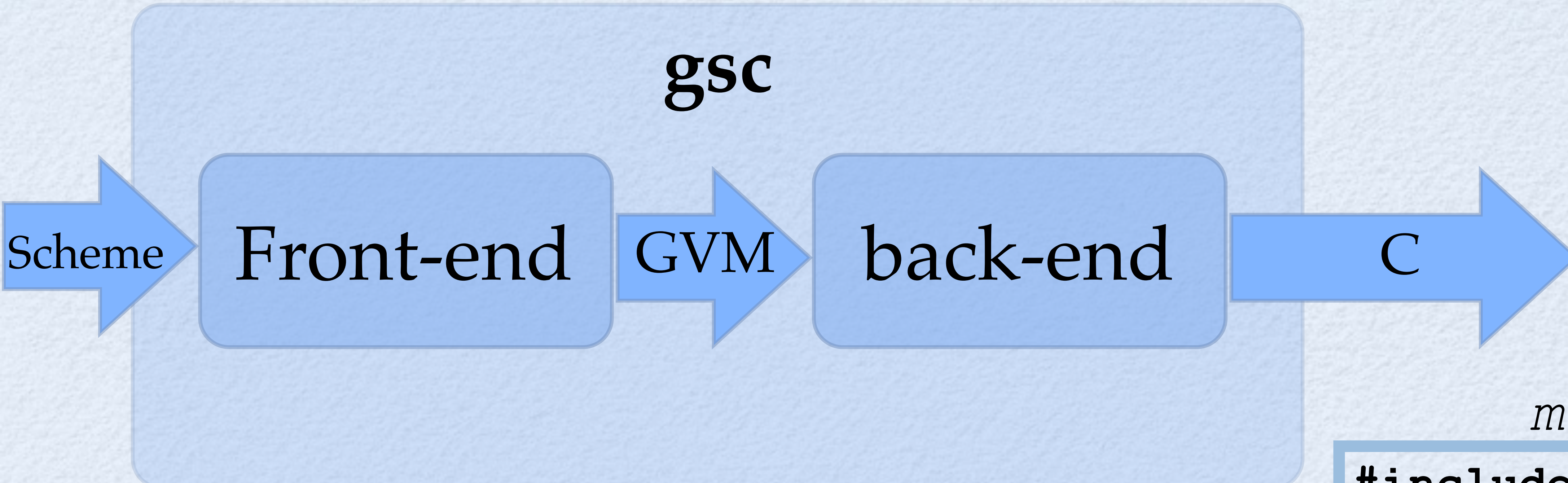
  simple to use + extensible + avoid arbitrary restrictions

- #2 - other nice features:
  - **Fast** (in the C ballpark or else I'll end up using C)
  - **Portable** (target platform should not matter)
  - **Reliable** (no bugs)
  - **Conformant** to standards when possible (to help build a community)

Design

- “**Bootstrap**” approach:
  - Write most of the system in Scheme and as few low level parts in C (Scheme compiler is good at enforcing important properties of the system and we prefer writing Scheme code!)
- 838 `.scm` files = 297,672 LOC
- 138 `.c`/`.h` files, LOC:
  - 80,435 hand-written
  - 1,221,788 generated by `gsc` from the `.scm` files

Gambit Virtual Machine



C backend example

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
```

mod1.c

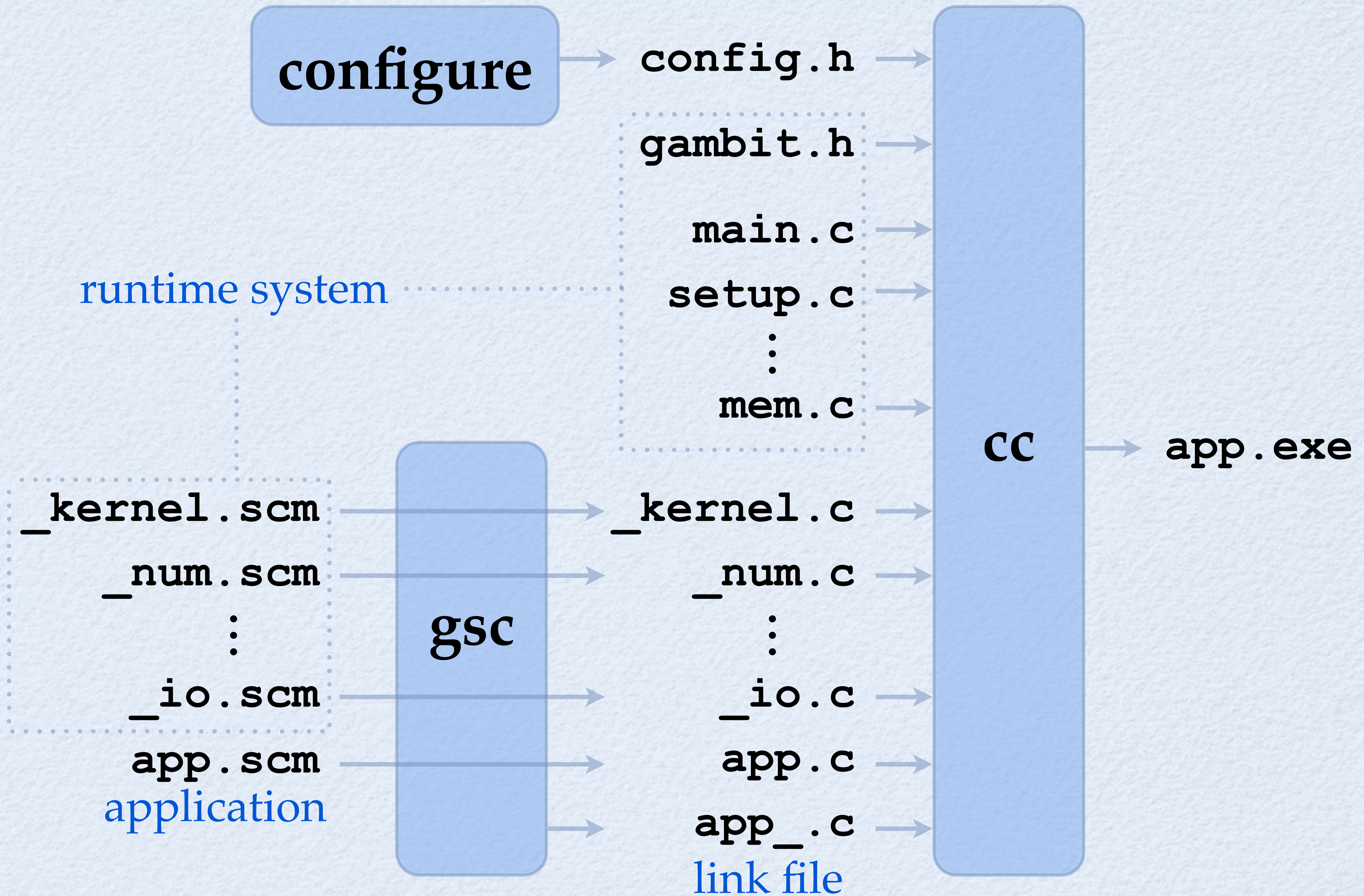
```
#include "gambit.h"

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))
   ...
```

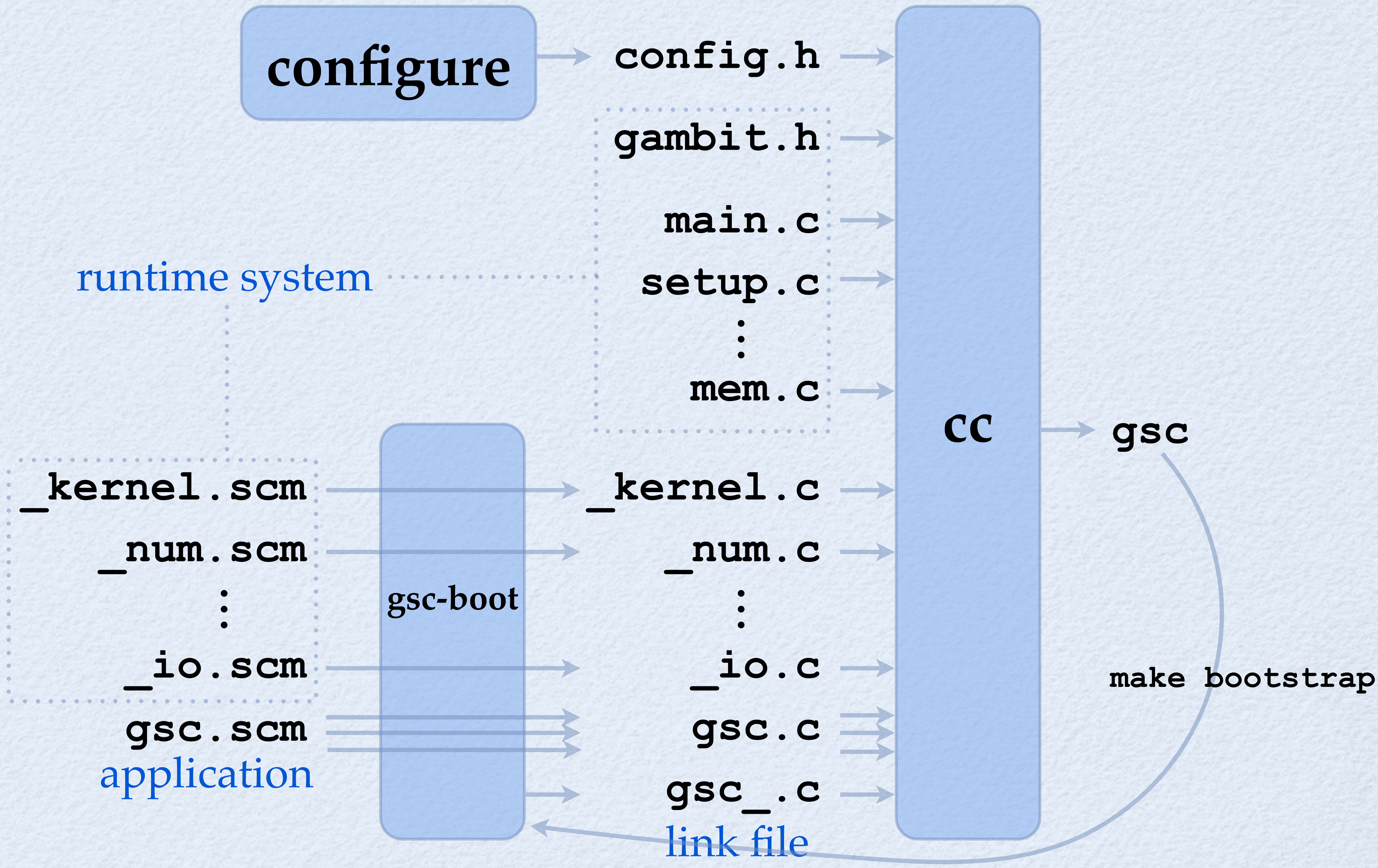
System Portability of C Backend

- **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
- Runtime depends only on basic 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



Gambit Bootstrapping



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)

RTS .c Source Code

```
 1164  main.c         main of all Gambit programs
 5865  setup.c        init/load/run compiled Scheme code
 7077  mem.c          memory management and GC
 6516  c_intf.c       FFI (foreign function interface)
  492  actlog.c       activity logs for profiling

 1936  os_base.c      basic IO, errors, mem alloc
 1090  os_dyn.c       dynamic code loading
 2491  os_files.c     filesystem operations
11612  os_io.c        IO subsystem (all types of ports)
 2985  os_setup.c     OS specific ops (pids, rusage, etc)
 1104  os_shell.c     getenv/setenv/system/...
  610  os_thread.c    abstraction of OS threads
 1676  os_time.c      time management
 8871  os_tty.c       Scheme friendly "readline"
------
53489 LOC
```

RTS .scm Source Code

```
  5417  _kernel.scm    init and intf to OS
  4350  _system.scm    equal, serialization
 12043  _num.scm       numbers and bignums
  2436  _std.scm       most standard procs
  5504  _eval.scm      expression evaluator
  1057  _module.scm    support for modules
 15326  _io.scm        IO, read, write
  3877  _nonstd.scm    non-standard ops
  8776  _thread.scm    (green) threads
  4589  _repl.scm      REPL and debugging
------
 63375 LOC (generates 481999 C LOC)
```

Compiler .scm Source Code

```
 1024  _host.scm        portability layer
 1434  _utils.scm       miscellaneous utilities
  591  _parms.scm       definition of common symbols
  327  _env.scm         compile time environments
 1635  _source.scm      files -> S-expr (deprecated)
 2816  _ptree1.scm      S-expr -> AST (parse trees)
 2990  _ptree2.scm      AST -> AST transformations
 5513  _prims.scm       AST -> AST transformations
 4235  _front.scm       frontend (AST -> GVM)
 2754  _gvm.scm         GVM -> GVM transformations
  632  _back.scm        interface to backends
 7083  _t-c-*.scm       C backend
12597  _t-cpu-*.scm     Native backend (x86, ...)
18594  _t-univ-*.scm    Universal backend (JS, ...)
  712  _gsclib.scm      compile-file, etc
------
62937 LOC (generates 615401 C LOC)
```

Recent Progress

- Close to full R7RS conformance
- Module system which simplifies code sharing and distribution
- A growing set of builtin modules (srfis, etc)
- Improvements to the native and universal backends
- Gambit on bare metal
- Geiser support (emacs plugging for Scheme dev)
- Benchmarking tool (Gambit forensics)

Schedule

- Development and history of Gambit (Marc Feeley) — done!
- Gambit forensics benchmark viewer (Sacha Morin)
- Gambit Virtual Machine (Marc Feeley)

- Gamboling with Gambit (Brad Lucier)
- Native backends (Abdelhakim Qbaich and Laurent Huberdeau)
- Gambit on bare metal (Samuel Yvon)
- Gambit geiser (Mathieu Perron)
- Yownu (Guillaume Cartier)

- Universal Backend and Migration (Marc Feeley)
- Gambit module system (Frédéric Hamel)

- Gerbil (Dimitris Vyzovitis)
- Together, JazzScheme (Barbara Samson and Guillaume Cartier)
- Compiler code walkthrough (Marc Feeley)