Selfie and the Basics

Christoph M. Kirsch
Department of Computer Sciences
University of Salzburg
Austria
ck@cs.uni-salzburg.at

Abstract

Imagine a world in which virtually everyone at least intuitively understands the fundamental principles of information and computation. However, computer science, in research and in education, is still a young field compared to others and lacks maturity despite the enormous demand created by information technology. To address the problem we would like to encourage everyone in the computer science community to go back to their favorite topic and identify the absolute basics that they feel are essential for understanding the topic. We present here our experience in trying to do just that with programming languages and runtime systems as our favorite topic. We argue that understanding the construction of their semantics and the self-referentiality involved in that is essential for understanding computer science. We have developed selfie, a tiny self-compiling C compiler, self-executing MIPS emulator, and self-hosting MIPS hypervisor all implemented in a single, self-contained file using a tiny subset of C. Selfie has become the foundation of our classes on the design and implementation of programming languages and runtime systems. Teaching selfie has also helped us identify some of the absolute basics that we feel are essential for understanding computer science in general.

CCS Concepts • **Applied computing → Education**; • **Software and its engineering** → *Compilers*; *Interpreters*; *Virtual machines*;

Keywords Computer Science for All, Self-Referentiality

ACM Reference Format:
Christoph M. Kirsch. 2017. Selfie and the Basics. In *Proceedings of 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!'17).* ACM, New York, NY, USA, 16 pages.
https://doi.org/10.1145/3133850.3133857

1 Introduction

This is a very personal piece. It is about how I hope computer science could be taught to more students, including the ones not majoring in computer science, and eventually even school children. What I describe here reflects how I learned about computer science and how I understand it ever since I started coding 35 years ago. But it is also about how I see things more clearly now after working in automated theorem proving, model checking, embedded and real-time systems, and concurrency and memory management. Nevertheless, I am neither a formal methods nor a systems person, and certainly not a programming language expert. This is important because I do a lot of cherry picking from all these fields while dismissing everything else potentially stepping on everyone's toes in the end. I like formal methods but I do not like complexity, especially in teaching. I really like engineering but only principled. The best, I think, is to do formal methods and principled engineering at the same time. So, the story goes like this.

Three years ago a colleague of mine at UC Berkeley asked me if I was interested in visiting him and teach computer science to graduate students for one semester in the civil engineering department. The idea was to have a computer scientist expose engineering students to a range of basic computer science topics. I gladly accepted the offer, prepared the class, and began teaching around 20 students in the Fall of 2014. It only took one week of teaching until I realized that my expectations on prior knowledge exceeded reality by far. I had to change my plans completely.

It turned out that the students were all coding every day without adequate formal training. I asked the students how they do it. The answer was: stackoverflow.com! They were following a simple development process: 1. search the Internet for answers and code, 2. copy-paste code that appears to be reasonable, 3. adapt and integrate that code until it "works". Most students appeared to me as quite unhappy about that choice but they nevertheless did not see any alternatives. There appeared to be little hope to gain the level of knowledge and confidence they were used to in their actual field of study.

That experience forced me to go back to the material I had been teaching for ten years and simplify it to the most basic level possible. This article is about that experience and its outcome. The most striking result is that being forced to be truly basic rather than comprehensive in teaching computer science does not only produce new non-trivial insight even into long established material but also reveals that very few absolute basics are sufficient and still get us surprisingly far.

The key observation with the engineering students was that they only had a vague understanding of the software they were developing even though they were all very bright and highly motivated. Sure, some of the software included non-trivial concurrent, distributed, and even real-time code but the problem was there even with sequential code. So, the question was and still is how to teach students computer science within a reasonable amount of time that leaves enough room for other fields of study. This question is of course not new and there have been many attempts at answering it which I discuss below.

For our attempt, we decided to go through my personal teaching material on systems engineering (architecture, compilers, operating systems, virtual machines) and theory of computation (logic, formal languages, automata theory, computability, algorithmic complexity) to see if there is a more or less linear path of absolute basics that the students need to know to understand the software they are developing. Identifying these basics turned out to be a lot more difficult than I thought and made me rethink how I teach all my classes, in particular my compiler and operating systems classes.

Eventually, I found myself developing an open-source software system called selfie[1] which helps me answer the question that I believe plays a key role in understanding computer science: How is the semantics of a formalism created by a machine and in particular how is self-referentiality involved in that process ultimately resolved?

Selfie helps answering that question using three well-established techniques, that is, compilation, interpretation, and virtualization. In fact, selfie is a 7k-line C program written in a single, self-contained file implementing:

1. a self-compiling compiler called starc that compiles a tiny but powerful subset of C called C Star (C*) to a tiny but powerful subset of MIPS32 called MIPSter,
2. a self-executing emulator called mipster that executes MIPSter code including itself when compiled with starc (I get to why self-execution is important later),
3. a self-hosting hypervisor called hypster that virtualizes mipster machines which can host all of selfie, that is, starc, mipster, and hypster itself, and
4. a tiny C* library called libcstar utilized by selfie.

There is also a simple linker, disassembler, profiler, and debugger as well as minimal operating system support in the form of MIPS32 o32 system calls built into the emulator. C* features only five different statements (assignment, while, if, procedure call, return) and has no data types other than signed integer and pointer to signed integer. The only way to access dynamically allocated memory is through the dereferencing operator * hence the name. Selfie runs in a terminal supporting any combination of self-compilation, self-execution, and self-hosting in a single invocation.

[1] http://selfie.cs.uni-salzburg.at

Two bachelor students have recently ported selfie to RISC-V in such a way that starc generates ELF binaries that are compatible with the official RISC-V toolchain, in particular the spike simulator and the pk kernel. Support of additional features is still in progress. For example, work on x86 support as well as on integrating a self-verifying theorem prover has just begun with a naïve SAT solver already in the code repository. More on that below.

Explaining selfie in class has lead me to develop a list of absolute basics that in my experience appear to be essential for understanding not just selfie but computer science in general. For example, knowing how basic information is encoded in bits or what the difference between compile time and runtime is appears to be essential for everyone trying to code. However, I have no scientific evidence of that of course hence the venue.

In the following I discuss the scientific and educational context of selfie and then present an overview of the system and what it can do in a how-to section. After that I present selfie in more detail. The selfie section and the how-to section are independent and may be read in any order. I then go through a Q&A session inspired by discussions I already had with colleagues and students. The Q&A may answer questions you might have but requires reading the selfie section first. Finally, I go through the list of absolute basics that I think are essential in teaching selfie and computer science in general. That section is again kept as independent as possible even though it is derived from teaching selfie. I provide a brief summary and an outlook on ongoing and future work at the end of this article.

2 Context

Selfie is inspired by seminal work in computer science and motivated by the challenge of teaching computer science to broader audiences than computer science majors. The list of textbooks I mention here is by no means complete. It only represents my favorite titles on the book shelf in my office.

2.1 Inspiration

The design of selfie is inspired by numerous projects and textbooks of which some go way back and are now considered classics in computer science. Selfie can be seen as an attempt to cherry pick the absolute basics from that material and combine it into a new self-contained system. In particular, there is seminal work related to selfie in systems [8, 14, 24, 28], theory [27], algorithms [20, 31], and programming [30] as well as architecture [15], compilers [32], operating systems [23] and programming languages [18, 25].

Selfie is written in C* which is a tiny subset of C [18] where all data is handled at the granularity of machine words similar to the programming language BCPL [25], an early predecessor of C. The self-compiling starc compiler in selfie is a single-pass, recursive-descent compiler similar to the Oberon compiler [32] but even simpler using, for example, a stack allocator for register allocation. Understanding the algorithms [31] and programming techniques [30] in selfie only requires the absolute basics of "The Art of Computer Programming" [20]. While the mipster emulator in selfie implements a tiny subset of MIPS [15] that is much simpler than most instruction sets, the hypster hypervisor is trickier. It is essentially a tiny microkernel [23] that virtualizes the machine emulated by mipster and, interestingly, shares most of its code with mipster.

Selfie's unique contribution is to demonstrate the fully self-contained construction of programming language semantics [14] from first principles [24] using self-referential compilation [32], interpretation [28], and virtualization [8].

Ongoing work on integrating a self-verifying theorem prover [21] into selfie is inspired by the success [29] of SAT and SMT solvers [5] as well as bounded model checking [4, 5], symbolic execution [6], whitebox fuzzing [13], and inductive theorem proving [17]. The goal is to provide, in addition to compilation, interpretation, and virtualization, another perspective on how to construct semantics, and eventually teach that perspective earlier than graduate school.

2.2 Teaching

There are numerous projects world-wide on teaching computer science to broader audiences than just computer science majors. The goal is widely seen as important for the global economy and society in general and has recently received increasing recognition, for example, by the US government[2] and funding agencies such as the National Science Foundation.[3] There is a similar trend in many other countries around the world such as the UK,[4] for example.

Selfie is related to projects that aim at identifying essential topics and principles rather than at developing pedagogical tools and platforms. While services such as Khan Academy, Coursera, and Udacity, to name a few, are important to reach broader audiences the long-term goal here is to identify the essential knowledge for understanding basic computer science and coding in particular. Projects such as the K-12 Computer Science Framework,[5] Bootstrap,[6] Code.org,[7] Program by Design,[8] and the Computer Science Field Guide[9] are large-scale efforts closer in spirit [11] based on extensive teaching experience and class evaluations.[10]

[2]http://wh.gov/blog/2016/01/30/computer-science-all
[3]http://nsf.gov/csforall
[4]http://www.computingatschool.org.uk
[5]http://k12cs.org/
[6]http://bootstrapworld.org
[7]http://code.org
[8]http://programbydesign.org
[9]http://csfieldguide.org.nz
[10]http://www.acm.org/education/CS2013-final-report.pdf

2.3 Target Audience

Selfie's target audience is undergraduate and graduate students in computer science and other engineering disciplines. Unsurprisingly, there is controversy among my colleagues and reviewers if a bottom-up approach from first principles is what non-computer-science students want and need. I can only say that the civil engineering students in the class I taught at Berkeley very much appreciated seeing how things really work. The only challenge was to find a way to simplify everything to the absolute basics while still being sufficiently realistic. This is a lot of work but in my opinion at the heart of the problem when it comes to reaching out to broader audiences, and it is something only the scientific community can do. Try it!

I even believe that, if not selfie, then at least the basics derived from teaching selfie can also be taught to school children, first playfully and later more formally. It ought to be possible to develop, in a younger audience, an intuitive understanding of what it means to automate problem solving. I still remember the first time I realized as teenager how a computer requires me to spell out everything for it to be able to solve anything interesting. There was no magic, just bits, and that was good. Nevertheless, after all these years I still only have experience in teaching students at college level.

3 How-to

The best introduction to selfie, I think, is to see how to use it. Bootstrapping selfie works on Linux, macOS, and Windows machines with minimal effort.[11] A terminal and a C compiler that can generate 32-bit binaries (option `-m32`) will do:

```
> make
cc -w -O3 -m32 -D'main(a,b)=main(a,char**argv)' \
selfie.c -o selfie
```

By now we have an executable `selfie` compiled from `selfie.c` which implements all of selfie in a single file. Selfie without options responds with its usage pattern:

```
> ./selfie
usage: selfie { -c { source } | -o binary | \
-s assembly | -l binary | -sat dimacs } \
[ ( -m | -d | -y | -min | -mob ) size ... ]
```

The order in which the options are used matters. Compiling selfie with selfie is simple and takes seconds on my laptop (option `-c`):

```
> ./selfie -c selfie.c
this is selfie compiling selfie.c with starc
191548 characters read in 7576 lines and 1032 comments
with 105824(55.24%) characters in 31199 actual symbols
270 global variables, 311 procedures, 455 string
literals, 2094 calls, 809 assignments, 71 while,
627 if, 287 return, 129704 bytes generated with
30755 instructions and 6684 bytes of data
```

[11]https://github.com/cksystemsteaching/selfie/releases/tag/Onward17

The system provides a detailed profile of input and output which we use in class for teaching C* and MIPSter. However, the above invocation does not output any code. The compiler did generate code and data (initial values of global variables and string literals) but only stored the binary in memory.

Selfie outputs generated code and data into a file, here `selfie.m`, like so (option `-o`):

```
> ./selfie -c selfie.c -o selfie.m
this is selfie compiling selfie.c with starc
...
129704 bytes with 30755 instructions and \
6684 bytes of data written into selfie.m
```

Executing that binary is simple. Have selfie load the binary (option `-l`) and then create a mipster instance with, say, 1MB of physical memory to execute it (option `-m 1`):

```
> ./selfie -l selfie.m -m 1
129704 bytes with 30755 instructions and 6684 bytes \
of data loaded from selfie.m
this is selfie executing selfie.m with 1MB of \
physical memory on mipster
selfie.m: usage: selfie { -c { source } \
| -o binary | -s assembly | -l binary | -sat dimacs } \
[ ( -m | -d | -y | -min | -mob ) size ... ]
selfie.m exiting with exit code 0 and 0.00MB of \
mallocated memory
this is selfie terminating selfie.m with \
exit code 0 and 0.13MB of mapped memory
profile: total,max(ratio%)@addr,2max(ratio%)@addr, \
3max(ratio%)@addr
calls: 2254,913(40.65%)@0x321C,489(21.73%)@0x35DC, \
278(12.34%)@0x327C
loops: 167,137(82.64%)@0x4F34,30(17.98%)@0x174,0(0.00%)
loads: 17527,913(5.21%)@0x3230,489(2.79%)@0x35F0, \
278(1.58%)@0x3294
stores: 9758,913(9.36%)@0x3220,489(5.01%)@0x35E0, \
278(2.84%)@0x3280
```

Any console arguments to the right of the `-m 1` option (there are none here) are passed as arguments to the code executed by that mipster instance. Since we execute selfie and there are no arguments selfie responds with its usage pattern. However, since mipster is in charge now there is more information on what happened. For example, there was no memory allocated with `malloc` but out of the 1MB of physical memory there was 0.13MB actually used by the system, namely for loading the selfie binary into the mipster instance.

Also, the profiler reports that there were 2254 procedure calls in total, out of which 40.65% or 913 calls were the most calls done to the same procedure. The first MIPS instruction implementing that procedure is stored at memory address `0x321C`. The second and third most called procedures are also listed. Similar information is provided for number of while loop iterations as well as memory loads and stores. In my experience providing quantitative information on relevant aspects of code as well as on code execution helps students develop an intuition on what they are dealing with.

Which procedure is called 913 times even though selfie does not really do anything? Simple. Execute the code generated for selfie right after compiling selfie in the same invocation like so:

```
> ./selfie -c selfie.c -m 1
this is selfie compiling selfie.c with starc
...
this is selfie executing selfie.c with 1MB of \
physical memory on mipster
...
profile: total,max(ratio%)@addr(line#), \
2max(ratio%)@addr(line#),3max(ratio%)@addr(line#)
calls: 2254,913(40.65%)@0x321C(~1438), \
489(21.73%)@0x35DC(~1481),278(12.34%)@0x327C(~1444)
loops: 167,137(82.64%)@0x4F34(~1850), \
30(17.98%)@0x174(~230),0(0.00%)
loads: 17527,913(5.21%)@0x3230(~1438), \
489(2.79%)@0x35F0(~1481),278(1.58%)@0x3294(~1444)
stores: 9758,913(9.36%)@0x3220(~1438), \
489(5.01%)@0x35E0(~1481),278(2.84%)@0x3280(~1444)
```

This does exactly the same as before with the additional benefit that approximate source code line numbers are now provided by the profiler. The most called procedure is at around line 1438 in the `selfie.c` source code. The procedure is part of libcstar and involved in bitwise shifting and string handling. Connecting machine code back to the source code for which it was generated is very helpful for students.

How about feeding some console arguments into the mipster instance that executes the code we just generated? How about letting that code compile selfie again? This takes a few minutes on my laptop and requires a larger mipster instance with 2MB of physical memory:

```
> ./selfie -c selfie.c -m 2 -c selfie.c
this is selfie compiling selfie.c with starc
...
this is selfie executing selfie.c with 2MB of \
physical memory on mipster
selfie.c: this is selfie compiling selfie.c with starc
...
selfie.c exiting with exit code 0 and 1.37MB of \
mallocated memory
this is selfie terminating selfie.c with \
exit code 0 and 1.25MB of mapped memory
...
```

This time around the code actually allocated 1.37MB of memory using `malloc`. The total amount of physical memory needed to execute the mipster instance was 1.25MB, slightly less than the dynamically allocated memory of which some is never accessed. On-demand paging in mipster avoids allocating physical memory for that.

So, we can compile selfie with selfie and then execute the generated code to compile selfie again. Will the code generated the second time be the same as the code generated the first time? Let us output two binary files `selfie1.m` and `selfie2.m`:

```
> ./selfie -c selfie.c -o selfie1.m -m 2 -c selfie.c \
-o selfie2.m
this is selfie compiling selfie.c with starc
...
129704 bytes with 30755 instructions and 6684 bytes \
of data written into selfie1.m
this is selfie executing selfie1.m with 2MB of \
physical memory on mipster
selfie1.m: this is selfie compiling selfie.c with starc
...
selfie1.m: 129704 bytes with 30755 instructions and \
6684 bytes of data written into selfie2.m
selfie1.m exiting with exit code 0 and 1.37MB of \
mallocated memory
this is selfie terminating selfie1.m with \
exit code 0 and 1.25MB of mapped memory
...
```

and check:

```
> diff -s selfie1.m selfie2.m
Files selfie1.m and selfie2.m are identical
```

A key challenge in teaching selfie is to make students truly understand self-compilation. It requires teaching all basic principles discussed below but virtualization. Students who finally understand usually report deep intellectual satisfaction and often describe the experience as progress beyond anything they have done before in their intellectual lives. Seeing that encouraged me to pursue self-referentiality even further. So, let us continue and see where this goes. Selfie can also output MIPS assembly, here into a file called `selfie.s`, as follows (option `-s`):

```
> ./selfie -c selfie.c -s selfie.s
this is selfie compiling selfie.c with starc
...
1293223 characters of assembly with \
30755 instructions written into selfie.s
```

That assembly would also pass the above fixed-point check, of course. It looks like this:

```
> more selfie.s
0x0(~1): 0x24080007: addiu $t0,$zero,7
0x4(~1): 0x24094000: addiu $t1,$zero,16384
0x8(~1): 0x01090019: multu $t0,$t1
0xC(~1): 0x00004012: mflo $t0
0x10(~1): 0x00000000: nop
0x14(~1): 0x00000000: nop
0x18(~1): 0x25083AA4: addiu $t0,$t0,15012
0x1C(~1): 0x251C0000: addiu $gp,$t0,0
0x20(~1): 0x24080FFF: addiu $t0,$zero,4095
0x24(~1): 0x24094000: addiu $t1,$zero,16384
0x28(~1): 0x01090019: multu $t0,$t1
0x2C(~1): 0x00004012: mflo $t0
0x30(~1): 0x00000000: nop
0x34(~1): 0x00000000: nop
0x38(~1): 0x25083FFC: addiu $t0,$t0,16380
0x3C(~1): 0x8D1D0000: lw $sp,0($t0)
0x40(~1): 0x0C007804: jal 0x7804[0x1E010]
...
```

Each line lists exactly one MIPS instruction, in particular its address in memory, the approximate source code line number for which it was generated, and its 32-bit-binary and assembly representations.

The debugger of selfie, which is based on mipster, uses that format to output each executed instruction (option `-d 1`):

```
> ./selfie -c selfie.c -d 1
this is selfie compiling selfie.c with starc
...
this is selfie executing selfie.c with 1MB of \
physical memory on mipster
$pc=0x0(~1): 0x24080007: addiu $t0,$zero,7: \
$t0=0,$zero=0 -> $t0=7
$pc=0x4(~1): 0x24094000: addiu $t1,$zero,16384: \
$t1=0,$zero=0 -> $t1=16384
$pc=0x8(~1): 0x01090019: multu $t0,$t1: \
$t0=7,$t1=16384,$lo=0 -> $lo=114688
$pc=0xC(~1): 0x00004012: mflo $t0: \
$t0=7,$lo=114688 -> $t0=114688
...
selfie.c exiting with exit code 0 and 0.00MB of \
mallocated memory
this is selfie terminating selfie.c with \
exit code 0 and 0.13MB of mapped memory
...
```

The important piece of information here is that the debugger also outputs, for each executed instruction, the part of the machine state affected by the instruction and on which the instruction depends before executing it (left of `->`) and the affected part of the machine state after executing the instruction (right of `->`). For example, the first executed instruction `addiu $t0,$zero,7` overwrites the value of the `$t0` register and depends on the value of the `$zero` register (`$t0=0,$zero=0`) because it stores the result of adding the value of the `$zero` register and 7 in the `$t0` register (`$t0=7`). By just following these arrows it is possible to reconstruct the full state of the machine at any time during code execution. In class I connect how machine state evolves back to how program state develops. In particular, I show students how much smaller the state space of C* programs usually is compared to the state space of MIPSter machines executing the generated code. This helps me explain, independent of any application, why reasoning about correctness of programs, rather than machine code, is simpler.

Another component of selfie useful for testing and debugging is the linker which supports linking C* source code as it is compiled. Suppose we would like to compile and run a C* program `integer.c` that implements its own `main` procedure. It outputs the decimal number 85 in various ways by calling printing functions implemented by libcstar in `selfie.c` yet without using any `#include` directives since they are not supported. Compiling and running that program works as follows:

```
> ./selfie -c manuscript/code/integer.c selfie.c -m 1
this is selfie compiling manuscript/code/integer.c \
with starc
...
this is selfie compiling selfie.c with starc
warning in selfie.c in line 7571: redefinition of \
procedure main ignored
...
130332 bytes generated with 30881 instructions and \
6808 bytes of data
this is selfie executing manuscript/code/integer.c \
with 1MB of physical memory on mipster
85 in decimal:     85
'U' in ASCII:      85
"85" string:       85
85 in hexadecimal: 0x55
85 in octal:       00125
85 in binary:      1010101
manuscript/code/integer.c exiting with \
exit code 0 and 0.00MB of mallocated memory
this is selfie terminating manuscript/code/integer.c \
with exit code 0 and 0.13MB of mapped memory
...
```

Source code files are compiled and linked in the order in which they are listed as console arguments. The redefinition of the `main` procedure in `selfie.c` is ignored with a warning. Tolerating redefinitions even allows us to reimplement functionality in selfie without changing `selfie.c`.

Now, let us do something seemingly crazy. Let us compile selfie with selfie, then execute the generated code to compile selfie again, and then execute that generated code to compile selfie again. In other words, let us run a mipster instance on top of a mipster instance and self-compile three times:

```
> ./selfie -c selfie.c -m 4 -c selfie.c -m 2 -c selfie.c
```

First of all, this takes hours on my laptop. In contrast, just compiling selfie with selfie takes seconds. As mentioned before, compiling selfie with selfie and then executing the generated code to compile selfie again takes a few minutes.

Going from seconds to minutes is because of mipster executing the generated code. Going from minutes to hours is because of another mipster running on top of a mipster. In fact, the slowdown is exponential in the number of mipsters.

So, why would we do this? Because it shows the argueably simplest path to implementing a kernel that can execute multiple processes concurrently while completely ignoring temporal and spatial performance. Consider the following, slightly different invocation of selfie (using option `-y 2` instead of `-m 2`):

```
> ./selfie -c selfie.c -m 4 -c selfie.c -y 2 -c selfie.c
```

This time, instead of running another mipster on mipster, we run a hypster instance on mipster. Logically, hypster does exactly the same as mipster, but instead of interpreting the code, hypster virtualizes a MIPSter machine by asking the mipster on which it runs to execute the code on its behalf (through context switching isolated in virtual memory).

What will the execution time be? Just a few minutes rather than hours, that is, a little more than twice as much time as compiling selfie with selfie once. This is true even if we run another hypster on hypster. In fact, the only slowdown in the number of hypsters is through context switching and paging. Virtualization removes the exponential. There is more on that below.

Selfie's latest design allows switching between emulated and virtualized code execution at runtime. Given any MIPSter code, selfie can have mipster (on top of another mipster) execute that code for a while and then switch over to hypster to execute it virtualized for some more time and finally go back to mipster to execute it and so on. We recently realized that this should be possible and got it to work with a few modifications to the existing code that actually made the implementation even cleaner than before. This work is part of an ongoing effort of showing functional equivalence of emulation and virtualization in selfie. More on that below.

There is one more thing. We have recently begun integrating verification technology into selfie and released, as first component of this project, a naïve SAT solver called babysat which supports the DIMACS format for SAT formulae in clausal normal form (option `-sat`):

```
> ./selfie -sat manuscript/cnfs/rivest.cnf
this is selfie loading SAT instance rivest.cnf
7 clauses with 4 declared variables loaded from rivest.cnf
p cnf 4 7
2 3 -4 0
1 3 4 0
-1 2 4 0
-1 -2 3 0
-2 -3 4 0
-1 -3 -4 0
1 -2 -4 0
rivest.cnf is satisfiable with -1 -2 3 4
```

We are now working on a minimal high-performance SAT solver but plan to keep babysat for educational purposes. Again, more on that below.

4 Selfie

Selfie's main features are the C* programming language, the MIPSter instruction set, the libcstar library, the starc compiler, the mipster emulator, and the hypster hypervisor, introducing them here in that order. While everything is implemented in a single file the compiler code is independent of the emulator and hypervisor code, and could therefore be implemented separately. However, all code depends on the library code, and most of the emulator and hypervisor code is shared including the bootloader and virtual memory. Early versions of the system were in fact implemented separately until we realized that putting all source code into a single file as well as all executable code into a single binary enables selfie's unique bootstrapping capabilities. Also, the single-file implementation removes the need for a build system.

4.1 The C* Programming Language

C* is a tiny subset of C designed to trade-off code size, simplicity, and readability of selfie while providing interesting challenges for compilation and educational opportunities for extensions. In particular, C* is an attempt at identifying a subset in which a self-referential system like selfie can be implemented with the least amount of code that is still simple and easily readable. It took around two years of selfie development work to settle on its design. During development we were driven by the observation that whenever we added a language feature we needed code to compile it but could potentially save or improve code elsewhere including the emulator and hypervisor when using that feature. This means that if we now add or remove a feature the code is likely to either grow, get more complicated, or less readable. For example, adding structs requires quite a bit of compiler code but would not save much code elsewhere while removing string literals would save and simplify compiler code but make the rest of the code much less readable. In short, we were doing programming language design driven by self-referentiality.

The grammar of C* is LL(1) and has six keywords (`int`, `while`, `if`, `else`, `return`, `void`). C* features five statements (assignment, while loop, if-then-else, procedure call, and return) as well as five built-in functions that are sufficient to bootstrap selfie (`exit`, `malloc`, `open`, `read`, `write`). Notably, there is no `free` and no `close`. Procedures can have arguments, local variables, and return values. There are standard arithmetic operators (`+`, `-`, `*`, `/`, `%`) and comparison operators (==, !=, <, <=, >, >=) but no bitwise and no Boolean operators. Integer, character, and string literals are supported.

There are only two data types in C*, namely signed integer (`int`) and pointer to signed integer (`int*`). In particular, there are no composite data types for two important reasons. Firstly, showing that something like selfie can be implemented without composite data types has a profound effect on students. It makes students realize that data types in programming languages are not needed for expressiveness which is something that comes as a surprise to many students. Secondly, not supporting composite data types in C* allows me to ask students to implement their own (usually, arrays and structs) which in my experience appears to make students truly understand the concept and motivation of data types.

The only way to access dynamically allocated memory in C* is through the dereferencing operator `*` which gave C* its name. Dereferencing and pointer arithmetic provide full control over memory access and therefore play a key role in systems programming and in particular in selfie. The basic principle that is so important for students to understand is that an integer in selfie can also be an address and vice versa. In other words, main memory is not just storage, it is always also an address space that is in the same value domain as the storage modulo word alignment. In my experience, C* helps students develop awareness and understanding of the duality of address and storage, in addition to the duality of code and data.

4.2 The MIPSter Instruction Set

MIPSter is a tiny subset of MIPS32. There are only 16 MIPSter instructions (`nop`, `addu`, `subu`, `multu`, `divu`, `mfhi`, `mflo`, `slt`, `jr`, `syscall`, `addiu`, `lw`, `sw`, `beq`, `jal`, `j`). MIPSter reflects what is needed to implement C*. In particular, there are no bitwise instructions and no sub-word data transfer instructions.

The only issue is with signed integer literals in C* (32 bits) that are too large to fit as immediate values in MIPSter (16 bits). Such values are decomposed by the compiler into the largest numerals in base two that can still be loaded into registers and then reconstructed with the available instructions. MIPSter itself can be taught in around two weeks of classes even to audiences with little architecture background.

4.3 The libcstar Library

The implementation of selfie includes its own tiny C* library called libcstar for bit manipulation, string manipulation, ASCII to integer (`atoi`) and integer to ASCII (`itoa`) conversion, and printing of strings.

Bit manipulation is implemented with signed integer arithmetic which is non-trivial since avoiding integer overflows is necessary for correct bootstrapping. The code serves as an educational example for students to see what signed integer arithmetic does and in particular how two's complement works. An exercise for students is to implement bitwise C* operators and bitwise MIPSter instructions. Replacing bit manipulation functions in selfie with bitwise C* operators results in significant performance improvement.

4.4 The starc Compiler

Selfie originally began with separate implementations of a simple, self-compiling, single-pass, recursive-descent C compiler and a simple DLX (DeLuXe) emulator that was the target of the compiler. DLX is an educational instruction set derived from MIPS with a focus on logical functionality rather than architecture-specific details [26].

Only later we realized that there is a subset of MIPS instructions that we now call MIPSter which is almost as simple as DLX but in fact part of a realistic ISA. We decided to implement a MIPSter emulator and port the compiler to target MIPSter instead. That port of the compiler eventually became what we now call the starc compiler.

Nevertheless, similar implementations of such compilers and emulators have been done many times before. The difference to other projects, however, goes back to two key observations. Firstly, we decided to focus on code size, simplicity, readability, and self-compilation of the compiler while keeping the design of programming and machine language open. We kept asking if we needed a particular language feature or not. During development of the compiler we realized that we could get away with no composite data types in the language by using a coding convention with getters and setters instead. This simplified the compiler dramatically while creating new opportunities for teaching. Secondly, we realized that we can implement compiler and emulator in the same file in such a way that the emulator can even run on top of another instance of itself. We therefore call mipster self-executing. More on that below.

Also, the backend of the compiler and the frontend of the emulator now appear right next to each other in the code. This simplifies code maintenance and helps teaching side by side how to encode machine instructions in the compiler and decode machine code in the emulator. Moreover, since compiler and emulator are compiled into the same executable, selfie can compile C* programs, including itself, and immediately execute the generated MIPSter code even without writing that code into a file. Computing the fixed point of self-compiled selfie executables can therefore be done in a single invocation of selfie.

Even more importantly, however, implementing compiler and emulator in the same file significantly simplifies teaching how to bootstrap a compiler. For this purpose, we have implemented each built-in function of C* in two procedures appearing side by side in the code. The first procedure is invoked by the compiler to generate wrapper code that invokes a system call using the MIPS o32 convention. The second procedure actually implements the system call and is invoked by the emulator. We have applied the same technique in the implementation of microkernel functionality for the hypster hypervisor.

Self-referentiality of compiler and emulator makes the choice of output language of the compiler become part of the equation over the choice of input language and the code size, simplicity, and readability of the overall implementation. The result is two tiny, easy to teach yet realistic languages, C* and MIPSter, and the starc compiler as well as the mipster emulator in their current form.

The design of the starc compiler is originally inspired by the Oberon compiler of Niklaus Wirth [32]. In addition to the choice of programming and machine language and the degree of self-referentiality, the difference is that starc has been designed purely for educational purposes. Moreover, starc does not perform any optimizations, not even constant folding. Register allocation is also simpler based on a naïve stack allocator.

4.5 The mipster Emulator

The mipster emulator is an unoptimized interpreter of MIPSter code. An interesting twist in its implementation is that we use, for example, the + operator for signed integer addition in C* to implement the `addu` instruction for signed integer addition in mipster. That instruction, however, is generated by starc for implementing the + operator. Students realizing that connection and how bootstrapping breaks the seemingly unbreakable cycle of self-referentiality often appear to be almost grateful for finally understanding this. At this point, we anyway emphasize that we could avoid using the + operator and instead implement a full adder using nothing but Boolean operations, or go even further and simulate the whole circuit, and so on. This is the time when students also understand that it is ultimately the physics of the underlying machine that determines the semantics [22] and that we are assuming elementary arithmetic as given to avoid going there.

Besides interpreting MIPSter code, mipster can also be invoked to disassemble a given binary into MIPS assembly, and even provide for each machine instruction approximate line numbers of the source code for which the instruction was generated, if compilation and disassembling is done in the same selfie invocation. Moreover, mipster profiles procedure calls, loop iterations, and load and store operations, and reports for each category the top three machine code locations, again including their approximate source line numbers if available. Lastly, for debugging purposes, mipster can be invoked to output each executed machine instruction and its effect on the machine state.

The fact that mipster is self-executing, that is, can execute itself allows me to demonstrate to students how simple it is to implement a kernel that can execute multiple processes concurrently if temporal and spatial performance is irrelevant. Just have a mipster running on top of another mipster create multiple emulator instances, load possibly different code into each, and then execute them round-robin style, maybe just one instruction in each emulator at a time. That mipster is a kernel, no need for context switching and virtual memory! In fact, I ask students to do this within a week as part of an introductory homework assignment.

However, as mentioned before, compiling selfie on top of two mipsters takes a day whereas doing the same on top of just one mipster only takes minutes. The way out of this is of course to remove the second layer of interpretation through virtualization as discussed next.

4.6 The hypster Hypervisor

The most advanced yet still accessible part of selfie is the hypster hypervisor which provides full virtualization of the machine emulated by mipster. My goal with hypster is to provide simple yet realistic infrastructure that enables students to develop their own operating system kernels. There are many other projects with similar goals such as PintOS from Stanford University, for example, but are typically a lot more involved.

Similar to the synergy of starc and mipster, we leverage the synergy of starc, mipster, and hypster all implemented next to each other in the same file. Here, we again make two key observations that are relevant. Firstly, instead of exposing the details of any interrupt and memory hardware we implemented the required functionality as part of mipster to provide an easier-to-understand execution environment for handling machine contexts and virtual memory on top of mipster. Secondly, by paging not just virtual but also physical memory in mipster, the code implementing hypster can in fact be made part of the code that implements mipster. The only difference between mipster and hypster is whether to interpret code (mipster) or to context switch the machine on which we run to interpret code on our behalf (hypster). This works when running hypster on top of mipster and even on top of another hypster, any number of times. We therefore say that hypster is self-hosting, similar to recursive virtual machines [12].

Independently of whether we run just mipster, or hypster on top of mipster, or even hypster on top of itself, the same code loads and runs binaries, and handles exceptions, interrupts, and page faults. Surprisingly, still, both mipster and hypster are really implemented by the same code except at the point where we decide to either invoke the interpreter or context switch to the interpreter that executes us. Selfie demonstrates that running a binary by interpretation or by context switching is logically the same thing. This is an insight that in my opinion is the key to understanding that self-referentiality in kernels is the fundamental source of their complexity and therefore requires proper logical orthogonalization.

5 Q&A

Rather than trying to provide hard evidence on why selfie should be as it is and before going into the details of how to teach students selfie, I go through my best anwers so far to a list of questions that came up when presenting selfie and the basics to colleagues.

Question #1: *Why did you pick C and MIPS rather than a modern, type-safe, maybe even functional programming language and a modern ISA?*

The short answer is I picked C and MIPS because this is my background. The long answer is three-fold: (1) unoptimized compilation targeting a simple but still realistic ISA such as MIPS is straightforward for an imperative language such as C, and (2) evil implicit state in programs written in C* (global variables!) as well as (3) evil pointer arithmetic as the only way to create dynamically allocated memory layouts in C* makes students truly appreciate the wonders of modern, type-safe and possibly functional programming languages.

Maybe the even more relevant question is whether to expose students to low-level machine-related artifacts or not. I think we need to do both, bottom-up training à la selfie as well as top-down coding using an adequate programming language [10]. I only focus on the bottom-up approach here and try to show how far we can simplify it so that we do not need to avoid teaching it. C* is specifically designed to feature structured programming for code but not data. This creates a sufficiently interesting compilation challenge while leaving most of the work on how to manage state to the students (if you want arrays and structs implement them!).

This is my best answer so far but I am not saying that C* is the best way to do selfie. In fact, porting selfie to a functional programming language could be a very interesting and exciting project yet potentially in direct competition with the classics [28]. In the meantime, we have already ported selfie to RISC-V which is a modern ISA. There is also ongoing work on x86 support.

Question #2: *Why do you draw the line at Boolean logic and elementary arithmetic rather than going down to the level of circuits?*

I believe Boolean logic and elementary arithmetic is in fact the global minimum in intellectual challenge in the sense that going down to the level of circuits as well as going up to the level of more abstract machines makes things more difficult to understand. A MIPSter machine is hard to make any simpler while still keeping things realistic. It does make sense to explore other levels of abstraction and there are examples of that [24] but probably only for students majoring in computer science.

Question #3: *How do you know if selfie is sufficiently realistic and not too simplistic?*

Good question. The design avoids any performance optimizations but is still structurally close to a realistic system. The key difference to a real system is self-containment. Selfie is a sandbox that features everything students need to experiment with real implementations of programming language concepts and runtime features. The system can easily be extended and made faster by students as part of weekly homework assignments as well as larger projects. For example, support of arrays and many other language features can be done in one-week assignments. Integrating user-level runtime features such as a more realistic memory allocator or even a conservative garbage collector and kernel-level components such as process schedulers and locks can readily be done as well.

Question #4: *How can I integrate selfie into my own classes? Do you have any recommendations for that?*

I would of course be delighted if you would use selfie in your own classes. Selfie as a system may be a bit too tough for first-year students. However, C* as an introductory language and the code in libcstar are easily accessible even for inexperienced students. The starc compiler is ideal as a standalone system for introductory compiler classes. It is less suited for more advanced classes on compiler optimization since starc is single-pass. However, simple optimizations such as constant folding can easily be done. Similarly, the mipster emulator provides a nice foundation for an introductory architecture class. The combination of mipster with the hypster hypervisor may nevertheless serve as foundation of introductory as well as more advanced operating systems classes because mipster can easily be extended to provide hardware features calling for support by adequate non-trivial extensions of hypster.

Question #5: *Why do you think that virtualization is essential rather than other ways of constructing semantics such as computational logic, for example?*

I actually do not think that. I do think that virtualization plays an important role and it ought to be possible to teach broader audiences what it is and how it works. However, other ways of constructing semantics are important as well. We are in fact exploring ways to integrate testing and verification technology into selfie such as symbolic execution inspired by KLEE [6], whitebox fuzzing inspired by SAGE [13], and even simple (!) inductive theorem proving inspired by ACL2 [17] that will hopefully be able to check simple (!) statements on some selfie code, hopefully even itself, in the spirit of the project. This could help exposing students to higher levels of abstraction and create awareness of the logics of programming. We have already begun laying the groundwork by implementing a naïve SAT solver in selfie. There will eventually be more efficient versions of SAT and SMT solvers and maybe a bounded model checker.

Question #6: *Why do you leave out so many interesting features such as a file system and a network stack, for example?*

My best answer is lack of time and background. I am neither a file system nor a network stack person but I would be delighted to see someone implement support of that in selfie. Other educational operating systems projects already support some of that. For example, PintOS does implement a simple file system. It would also be nice to have a (self-referential?) terminal implemented in selfie.

Question #7: *Why do you use self-reference to define semantics rather than a formalism specifically designed for defining semantics?*

My short answer is that I aim at the arguably simplest yet still self-contained introduction to computer science. Self-referentiality allows me to do exactly that. There is no need to introduce any additional concepts and formalisms. We bootstrap from Boolean logic and elementary arithmetic.

My long answer goes like this. Programming languages are formalisms whose semantics can be defined in different ways. While defining semantics using mathematical notation specifically developed for this purpose is standard for computer scientists, it requires defining and understanding the semantics of that notation as well to be self-contained. Moreover, while semantics may be mathematically defined, it is ultimately the machines executing code that determine its behavior. I therefore believe that self-referentiality is a fundamental concept that is hard to avoid.

In fact, programming languages and runtime systems are by construction self-referential, just like natural languages, even though they are otherwise quite different concepts. Interestingly, teachers of foreign language classes often speak exclusively in the language they are teaching and use other means of communication including body language to explain semantics. Nevertheless, people speaking a language to communicate may or may not need to understand self-referentiality which is probably an interesting question studied by linguists and neuroscientists. However, programming languages are not natural languages. Their semantics is formally defined. We may not need to use the same programming language or even any programming language at all to define the semantics of a programming language but can anyway not avoid self-referentiality.

Not understanding self-referentiality in programming languages and in particular not knowing how self-referentiality is resolved means that our understanding of programming language semantics is not well-founded. This may or may not be fine for natural languages but it is definitely not for programming languages and in fact any notation with formal semantics. People learning about computer science need to know that programming can be a lot of fun but is ultimately a formal activity, no matter how casual the programming language they use may appear. Interestingly, even when reasoning about the limits of computation self-referentiality shows up, as in diagonalization proofs, for example. There are abstract mathematical statements such as Cantor's Theorem that provide great opportunities for creating further intuition on the construction of semantics.

Question #8: *Do you really believe that a system like selfie and the self-referentiality involved in it can help teach everyone computer science?*

Yes, I do believe that. Teaching programming languages informally is important and helps students learn how to code. However, even though teaching selfie and in particular self-referentiality may appear to be difficult does not mean that it should be avoided. For computer science and coding in particular to become an established discipline like other much older disciplines I do not see much of an alternative than to identify and teach principled material and simplify everything involved in that as much as possible. There is no reason why computer science needs to remain an often admired but still inaccessible subject in schools and colleges and even society in general around the world.

6 The Basics

Now with all the fun stuff taken care of the only thing that remains to be done is to figure out how to teach computer science to people who have no clue. Surprisingly, the solution is right there in that statement. Just imagine what it is like to explain something like selfie to someone who does not know anything about computer science (or worse, has false knowledge about computer science). Explaining selfie under that assumption has helped me gather a list of absolute basics I call "principles" that in my experience are essential in understanding selfie. In fact, you may be able to do the same in your favorite area of computer science. Build a representative system similar to selfie and then try to explain that system in order to identify the "principles" essential for understanding it.

In the meantime, the "principles" I present here have become the conceptual and structural foundation of a textbook on selfie that exists as an early draft online with three chapters finished and the fourth in progress.[12] The book introduces each "principle" in a dedicated chapter using selfie as motivation and example. Students are encouraged to read the book and work with the code of selfie at the same time in order to improve their understanding of each principle.

The idea of presenting and teaching a list of absolute basics differs from the categorization found in many textbooks. Unfortunately, I have no evidence of completeness or even just necessity. The list is really just the result of trying to explain selfie. What do you need to know to understand the compiler, emulator, and hypervisor? The answer is that list. I then go one step further and claim that understanding selfie is key to understanding computer science because selfie provides well-established and self-contained ways of constructing semantics on digital computers. That argument makes the list of absolute basics essential for teaching computer science.

Principle #1: *Semantics*

What is the meaning of the bits stored in a computer and where does it come from? The answer is there is none unless the machine does something with them. As is, bits have no meaning. The semantics principle is that change determines semantics. In particular, the meaning of bits comes from change over time, by being changed or by changing others, as determined by the machine storing and executing the bits. For example, bits represent data such as an integer value at the time when they are used as operand of an integer operation. Bits represent code such as an integer operation at the time when they direct the machine to perform an integer operation. At all other times, their meaning could be anything. This also means that bits can sometimes be data, code, or even both. Self-compilation exemplifies that phenomenon.

The semantics principle creates awareness of the fundamental difference between representation and meaning, and provides a simple explanation of the origin of semantics of formalisms such as source code and machine code. The earlier students and hopefully school children become aware of the semantics principle, first playfully and later more formally, the easier it may be for them to reflect about their own understanding of any kind of notation.

In class, just showing how to build and run selfie provides a number of great opportunities to explain the semantics principle. In particular, I point out that selfie is implemented by a sequence of characters encoded by a sequence of bits. Compiling selfie translates these bits into another sequence of bits encoding the executable machine code which we then run on a machine. That machine code can of course be executed to translate the source code again and so on. Learning about self-referentiality early has two interesting effects on students: confusion and curiosity. I leverage the confusion by trying to get students with prior but incomplete and often false knowledge to question their knowledge, and all students to reflect on the difference between representation and meaning.

Principle #2: *Encoding*

But how is code and data actually encoded in bits and why is it done in one way over another? Depending on how information is encoded, computing with it may be faster or slower and storing it may require more or less space. However, encoding does not create any meaning. For example, the bit string 1010101 may represent the decimal number 85 but also the uppercase ASCII letter U or in fact any other information. However, the binary encoding of an integer such as 85 is exponentially more succinct than its numerical value (just 7 bits rather than 85) enabling compact storage while arithmetic operations are still fast thus making it the default for encoding integers.

The encoding principle is to distinguish how information is encoded, bounding temporal and spatial performance of computation, from how to compute with it. The encoding principle creates awareness of the importance of representation and its impact on temporal and spatial performance when handling it. Teaching students and school children how to encode numbers, for example, using other notation than decimal is simple and can be done before even talking about computation.

In class, we study how C* literals including characters and integers but also MIPSter instructions are encoded in bits. Surprisingly, even more experienced students often do not know how this is done and certainly not why it is done

[12] http://leanpub.com/selfie

the way it is. Many students may have not even realized the exponential difference in size between unary and binary encodings of numbers while going from binary to decimal encoding does not make much of a difference. Explaining the encoding principle is another great opportunity to point out that an encoding as is still does not have meaning but only facilitates creating meaning during computation.

Principle #3: *State*

Ok, but what is computation really? For example, the process of encoding a decimal number in bits is in fact computation which works by accumulating the numerical value of that number incrementally one digit at a time. The state of this computation is the accumulated value and, not to forget, the control state of the code that implements the encoding. The state principle is to model computation as the evolution of state, that is, as a sequence of states encoded in bits. Each step in a computation determines the change in bits from one state to the next. A correct program keeps computation in good states whereas a bug may take it to a bad state.

The state principle reveals the importance of state during computation. In particular, it reveals that coding is about staying in good states. In combination with the encoding principle it also reveals the potentially enormous amount of states involved in a computation as the key challenge in engineering correct software. In my experience, many students do not reflect about the nature of computation on that level of abstraction even though it is simple to do. The consequence of that is a lack of understanding that there should be as little state as possible to make establishing correctness easier. I therefore stress in the next two principles that remembering the least and forgetting the soonest are essential principles in programming.

In class, I introduce C* and MIPSter code simultaneously and demonstrate how code execution creates the notion of program and machine state, respectively. In particular, I use examples of C* programs with a focus on simple numerical computation and their selfie-compiled implementations in MIPSter. Importantly, there is no dynamic memory management here, not even implicit stack allocation with procedures. This can be seen as an introduction to programming on two different levels of abstraction at the same time. It requires explaining, at least intuitively, semantics of C* without procedures and the von Neumann machine model, CPU and memory, and how they connect via state.

Principle #4: *Regularity*

Now, what state information do we really need to remember in a computation? The answer is that we should not focus so much on remembering information but rather on forgetting it. The regularity principle is to use finite state code whenever possible since it is regularly forgetful, that is, when the same state is computed twice everything is forgotten that happened in between. This is of course the essence of the proof of the pumping lemma for regular languages. The regularity principle focuses on the essential feature of finite state machines which is the capability to forget automatically eventually almost everything simply by proceeding in a computation. It creates awareness that finite state is the simplest form of memory management and that forgetting information on a machine can be a challenge. Kleene's theorem provides an alternative, more abstract language-theoretic perspective on the regularity principle.

Students are often not aware that their code is finite state and therefore does not require any form of dynamically allocated memory. Just being made aware of that is already an essential step towards understanding how to write correct code. Interestingly, however, students sometimes rightfully point out that any digital computer is by construction finite state. My response is that computer scientists consider the state space of a computer as unbounded anyway because the machine is too slow and the state space too large for full exploration in reasonable time.

In class, we study the regular portion of the C* grammar and work with the C* scanner as well as the command line scanner of selfie to explain the regularity principle. I ask students to extend the C* grammar and scanner to support new symbols for bitwise operators, for example. Students usually appreciate seeing the duality of regular expressions and finite state machines in practice. I even ask students to spot, in the source code, the states of the finite state machine implemented by the C* scanner. The natural next step is to show the parts of the C* grammar that cannot be handled by a finite state machine anymore. However, there is of course a simple way of handling unbounded state and still forget almost everything just by continuing in a computation.

Principle #5: *Stack*

What is the arguably simplest, safest, and most efficient form of managing unbounded state? It is stack allocation, explicitly, but also implicity using, for example, call stacks. The stack principle is to consider stack allocation, if finite state is not enough, before any other method for managing unbounded state. If forgetting state in reverse of remembering it is good enough, stack allocation should be used. Functional programming experts are probably not the only ones who would agree with that.

Procedures (in imperative languages) and functions (in functional languages), for example, are implicit means for stack allocation but rarely introduced as such. Students are often not aware of that point of view but like it because stacks are easier to understand than recursion, for example. But even before it comes to programming, just developing an intuition of what a stack is, how it works, and that it can do a lot more than one would expect is within reach of a broader audience.

In class, we introduce C* procedures at this point and show, by example, how they translate to MIPSter code and how that code constructs the call stack in memory. Also, we study the non-regular, context-free portion of the C* grammar and work with the C* parser to explain the stack principle. Similar to before, I ask students to extend the C* grammar and parser to support bitwise operators in arithmetic expressions, for example, to experience the duality of context-free grammars and pushdown automata in practice. Here, the obvious question is to ask students to spot, in the source code, the stack of the pushdown automaton implemented by the C* parser. Both the regularity and the stack principle emphasize the importance of forgetting rather than remembering state. Unfortunately, however, there is often more state to remember which often cannot be forgotten in reverse either.

Principle #6: *Name*

How do we compute with unbounded state that we have to remember and cannot forget in reverse? A hashtable whose content changes depending on program input is an example of such a scenario. Whatever the method is, however, managing unbounded state requires finite representation. The name principle is to distinguish, for each entity in a program, the name of the entity and the entity itself, at least in your mind. There is no anonymity here, that is, an entity without name does not exist. For example, as long as there is a reference to the hashtable it exists along with all the references to its content and the content itself.

There are of course countless other examples where the distinction between name and entity is important such as variable and procedure names in source code versus values and control state during execution, for example. However, the principle is always the same and creating awareness of the difference helps address many issues in managing unbounded state. Forgetting information in arbitrary order works by forgetting names in namespaces explicitly created for this purpose. Students with prior knowledge are often not aware that they generally need to say which information is still needed, or conversely, not needed anymore, even in managed languages (by destroying references). Ask your students what a reachable memory leak is!

In class, we use the symbol table of the starc compiler as example and show how it implements the scope of procedures and variables in C*. In fact, this is the first time the students see the use of malloc in selfie and learn that the heap is just a range of memory addresses, just like the stack. Here, I would like to emphasize the importance of exposing students to the conventions for using memory. Many students are surprised by how simple this is, one pointer for the top of the stack growing downwards towards the heap and one (bump) pointer for the end of the heap growing upwards towards the stack, that's it. Ultimately, however, keeping namespaces finite, that is, saying what is needed and what is not is required to manage unbounded state. This takes us to the next principle.

Principle #7: *Time*

What is the difference between compile time and runtime? It is countability. There is more behavior or in fact semantics than there is code. Philosophically speaking, we can program (enumerate) what we want but we cannot compute what we want. The time principle is to distinguish programming, the design of a finite piece of code, and computing, the potentially unbounded execution of code, again at least in your mind. For example, the difference between a variable declaration or procedure definition and the use of a variable or the invocation of a procedure must be clear, especially since there is usually no syntactic distinction in many programming languages.

In my experience, students often understand the difference only superficially, which is usually revealed when it comes to self-compilation. But even less ambitiously, students struggling with understanding recursion, for example, can often be helped by pointing out the difference between definition and use of procedures. But also more abstract content such as Cantor's Theorem, unlike Gödelisierung, for example, should be within reach of a broader and also younger audience. It is then up to us to point out how Cantor beautifully explains the difference in cardinality between programming and computing.

In class, we also look at code generation in the starc compiler to figure out the difference between compile time and runtime. Extending that part to support bitwise operators, for example, is often difficult for students because they need to understand which code the compiler needs to generate and how exactly this is done but also what happens when the generated code actually executes. Implementing support of simple code optimization techniques such as constant folding, for example, is helpful for students as well. In short, the challenge is to understand compilation and execution at the same time. Students are typically not used to thinking in multiple timelines. Using a system like selfie appears to help but the challenge remains, unsurprisingly, given the depth of the topic. Interestingly, there is also a spatial principle that is essential for understanding how selfie and digital memory in general works.

Principle #8: *Memory*

What is an essential property of digital memory? It is spatial contiguity. Digital memory not only provides storage for information but also address space. The memory principle is to distinguish contiguous and non-contiguous use of memory addresses for trading off temporal and spatial performance of memory management and access. Constant-time access of contiguous memory, assuming that arithmetic operations are constant-time, comes at the expense of memory fragmentation which may or may not manifest in increased memory consumption. Non-contiguous memory access may not be constant-time and require additional storage for referencing but may help avoid memory fragmentation through the use of same-sized blocks. Spatial locality and false sharing are consequences of spatial contiguity as well and can make orders-of-magnitude difference in performance.

The memory principle provides a great opportunity to teach students how to answer questions about whether they should use, say, a list or an array. Students realize how important it is to control the time and space it takes to address and store information in memory. Teaching how data structures are stored (and encoded) in memory or in fact any type of digital media, at least in principle, is essential because it reveals the fundamental difference between memory addresses and storage as well as managing and accessing memory, one of my favorite topics [1–3, 7].

In class, I ask students to implement support of arrays and structs in the starc compiler, and then port selfie to use arrays and structs rather than pointer arithmetic. Not just computing but also generating code that computes the correct addresses of array elements and struct fields in memory is an important exercise for students. Combining that with constant folding is helpful as well. Once students have implemented their own arrays and structs, the question comes up what the role of data types really is.

Principle #9: *Type*

Is there a way to program the intended meaning of some of the bits stored in memory? Yes, of course! The type principle is to distinguish intended semantics (the what) and actual semantics (the how) of code and data through typing. This is essential for developing a deeper understanding of how semantics is created on a machine. Typing introduces something as simple and purely syntactic as the name of a data type for an entity as complex and abstract as a value domain, for example, and then allows deriving semantical information from that without running the code.

Students are often used to programming with data types. However, many students are not aware that typing is an efficient and incredibly successful way to overapproximate, even at compile time, actual semantics which can in general not be determined without running the code. The type principle deepens the understanding of the other principles by showing that there are ways to construct abstract semantics that do not require the whole machinery of computation. Typing is thus another way to reduce the state space of computation, similar to forgetting information as soon as possible.

In class, I show students that the starc compiler does indeed perform simple type checking on signed integers and pointers to signed integers. While some of that is optional, type checking in expressions with pointer arithmetic is not. Students are then asked to implement their own, more advanced type checking on arrays and structs. At this point, students realize that they make the compiler "understand" the intention of a data structure, provided the data structure is properly typed and not just accessed through pointer arithmetic. From what I hear many students seem to like that insight very much.

Principle #10: *Bootstrapping*

Now, how do we finally break that apparent cycle of determining the semantics of bits using bits? The bootstrapping principle is to distinguish bootstrapping computation and actual computation, acknowledging that any computational system is incomplete in the sense that it requires support from outside the system to start up. Knowing how to bootstrap something as practical as a compiler or something as abstract as the definition of a formalism is essential for understanding how semantics is created from first principles.

In my experience, many students do nevertheless not even know how a computer actually boots and how the code they develop is eventually loaded and executed on their machines. I see this as yet another challenge in teaching computer science and believe that there should always be an explicit focus on how to bootstrap any non-trivial system whatever the actual topic is.

In class, I discuss how a computer boots but also how the mipster emulator in selfie boots, that is, how selfie loads MIPSter code into a mipster instance and then starts the emulator to execute it. The boot loader in selfie is important for students to see because it shows how a machine, here the mipster instance, requires external help, here selfie, to begin its existence. Interestingly, the boot loader is exactly the same even for hypster instances.

The principles presented so far are derived from the process of creating semantics through translation. The final two principles provide another essential perspective on creating semantics through interpretation and virtualization as done by most runtime systems.

Principle #11: *Interpretation*

What is the simplest way to implement the essence of an operating system kernel that can run n binaries concurrently and spatially isolated from each other on the machine on which the kernel runs? The answer is to have the kernel maintain n interpreters, one for each binary, and have the interpreters in round-robin fashion execute, say, one instruction at a time. To my own initial surprise, there is no need for processor context switching and virtual memory management at all, if we completely ignore temporal and spatial performance for a moment and just focus on what an operating system does logically. However, interpreted code is of course slower than native code by a significant constant factor, which increases exponentially with the number of interpreters (meaning operating systems) running on top of each other. Virtualization avoids that as we see below.

The interpretation principle is that interpretation defines semantics in terms of the formalism in which the interpreter is written but is exponential time in the number of interpreters running on top of each other. The interpretation principle shows that teaching even advanced behavior such as concurrent code execution is within reach and explains why it is anyway worth replacing interpretation with native code execution through virtualization. In my experience, students appreciate this point of view and report that it makes them understand, often for the first time, what operating systems and other runtime systems actually do.

In class, we exploit the self-referentiality of the mipster emulator to explain and demonstrate the interpretation principle firsthand. Support of running *n* binaries concurrently through interpretation can easily be done in a one-week assignment. Once this is done, we take the opportunity to design and implement other well-known concepts such as shared memory, atomic instructions, locks, and so on. In more advanced classes we then ask students to experiment in selfie with lock-free concurrent data structures, another favorite topic of mine [9, 16].

Principle #12: *Virtualization*

How can we encapsulate interpretation as well as context switching and memory isolation such that an operating system kernel can execute user code either by interpretation or natively using essentially the same kernel infrastructure? Upon executing user code, the kernel would either interpret that code or ask the processor on which it runs to execute that code natively in its own address space isolated from the kernel's address space. This does indeed work in selfie and shows that teaching even virtualized concurrent code execution is within reach if functionality and performance are properly orthogonalized.

The virtualization principle is that virtualization defines semantics in terms of the semantics of the virtualized system and that the time to execute virtualized code is equal to the time to execute the code unvirtualized plus the time to execute the virtualizing code. The virtualization principle reveals the essence of virtualization, and also its weakness which is the fact that the performance of virtualization approaches that of interpretation if the virtualizing code has to run too often, that is, in the limit for each instruction of the virtualized code. Introducing virtualization as a form of interpretation that is often efficient in practice is much appreciated by many students. In essence, I am saying that in order to understand virtualization we first need to build our own version of what we want to virtualize through naïve but functionally equivalent interpretation and only then virtualize that.

In class, we do exactly that. Each new feature is first implemented through interpretation and only then virtualized with the hypster hypervisor. Imagine all operating systems came with a simple (!) but functionally equivalent interpreter-based version as executable specification of what they are supposed to be doing. That would open up a lot of interesting directions in teaching and already does in research [19].

7 Conclusions and Future Work

How can we teach computer science to broader audiences than just computer science students? Is asking how to create the semantics of a formalism on a machine the right question? In order to find out we have developed selfie, a self-compiling compiler of a tiny subset of C, a self-executing MIPS emulator targeted by the compiler, and a self-hosting hypervisor of the machine emulated by the emulator. Selfie is a self-contained 7k-line C implementation in a single file. Selfie exemplifies how to create semantics on a machine using compilation, interpretation, and virtualization.

I use selfie in all my compiler and operating systems classes, and plan to include other classes such as a software verification class later. In fact, selfie may eventually help me develop a new class structure in which I do not distinguish between classes on compilers, operating systems, theorem provers, and so on but instead categorize according to concepts and properties: computer science classes on semantics, concurrency, logics, performance, and so on. This is something I am still working on and probably another interesting topic for debate.

We are currently exploring formal verification based on SAT and SMT solving as well as bounded model checking and inductive theorem proving to provide another perspective on how to construct semantics in future versions of selfie. The goal is to introduce formal methods into our bachelor curriculum and study ways to leverage the synergy of verification techniques within selfie. For example, an interesting long-term effort is to verify the functional equivalence of the mipster emulator and hypster hypervisor.

Acknowledgments

This work has been supported by the National Research Network RiSE on Rigorous Systems Engineering (Austrian Science Fund (FWF) Grant S11404-N23 and S11411-N23), a Google PhD Fellowship, and a Google Research Grant.

Selfie is the result of more than ten years of studying and teaching systems engineering. Many colleagues and students have been involved in its development through inspiring conversations as well as numerous coding sessions. In particular, I thank Alireza Abyaneh, Martin Aigner, Sebastian Arming, Christian Barthel, Armin Biere, Heidi Graf, Andreas Haas, Thomas Hütter, Michael Lippautz, Cornelia Mayer, Simone Oblasser, Sarah Sallinger, Raja Sengupta, and Ana Sokolova as well as all the students who have taken my compiler and operating systems classes over the years. I also thank James Noble as well as the other anonymous reviewers for their constructive feedback on earlier versions of this paper. The design of the starc compiler is inspired by the Oberon compiler of Professor Niklaus Wirth from ETH Zurich [32]. The design of the microkernel in hypster is inspired by microkernels of Professor Jochen Liedtke from University of Karlsruhe [23].

The idea of teaching the absolute basics of computer science came up in the Fall of 2014 in conversations with Professor Raja Sengupta at UC Berkeley while teaching graduate students in the systems program of Berkeley's Civil and Environmental Engineering Department. We noticed that most students spend considerable time on coding, independent of their backgrounds and interests, yet with little understanding of some of the most basic principles in computer science. That class turned out to be a revelation to us. Breaking down computer science into the absolute basics appears to be exactly what students want and need.

Selfie has already enabled me to unify my compiler and operating systems classes, and inspired local colleagues to use selfie in teaching their classes. The project has also inspired me to start writing my first textbook. I owe a particular thank you to the class of 2014 at Berkeley that endured our experiment which eventually lead to the development of selfie. I am particularly proud of one student who even changed his major to computer science after taking that class, eventually got his masters in computer science, and by now works as software engineer at Google Inc.

References

[1] M. Aigner, A. Haas, C.M. Kirsch, M. Lippautz, A. Sokolova, S. Stroka, and A. Unterweger. 2011. Short-term Memory for Self-collecting Mutators. In *Proc. International Symposium on Memory Management (ISMM)*. ACM, 99–108. https://doi.org/10.1145/2076022.1993493

[2] M. Aigner and C.M. Kirsch. 2013. ACDC: Towards a Universal Mutator for Benchmarking Heap Management Systems. In *Proc. International Symposium on Memory Management (ISMM)*. ACM, 75–84. https://doi.org/10.1145/2555670.2464161

[3] M. Aigner, C.M. Kirsch, M. Lippautz, and A. Sokolova. 2015. Fast, Multicore-Scalable, Low-Fragmentation Memory Allocation through Large Virtual Memory and Global Data Structures. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 451–469. https://doi.org/10.1145/2814270.2814294

[4] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. 1999. Symbolic Model Checking without BDDs. In *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (LNCS)*, Vol. 1579. Springer, 193–207.

[5] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (Eds.). 2009. *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, Vol. 185. IOS Press.

[6] C. Cadar, D. Dunbar, and D. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proc. USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 209–224.

[7] S.S. Craciunas, C.M. Kirsch, H. Payer, A. Sokolova, H. Stadler, and R. Staudinger. 2008. A Compacting Real-Time Memory Management System. In *Proc. USENIX Annual Technical Conference*.

[8] E.W. Dijkstra. 1968. The Structure of the "THE"-Multiprogramming System. *Commun. ACM* 11, 5 (May 1968), 341–346. https://doi.org/10.1145/363095.363143

[9] M. Dodds, A. Haas, and C.M. Kirsch. 2015. A Scalable, Correct Time-Stamped Stack. In *Proc. Symposium on Principles of Programming Languages (POPL)*. ACM, 233–246. https://doi.org/10.1145/2676726.2676963

[10] Matthias Felleisen, Conrad Barski, and David Van Horn. 2013. *Realm of Racket: Learn to Program, One Game at a Time!* No Starch Press.

[11] M. Felleisen and S. Krishnamurthi. 2009. Viewpoint: Why Computer Science Doesn't Matter. *Commun. ACM* 52, 7 (July 2009), 37–40. https://doi.org/10.1145/1538788.1538803

[12] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. 1996. Microkernels Meet Recursive Virtual Machines. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*. ACM, 137–151. https://doi.org/10.1145/238721.238769

[13] P. Godefroid, M. Y. Levin, and D. Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proc. Symposium on Network and Distributed Systems Security (NDSS)*. 151–166.

[14] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison Wesley.

[15] John L. Hennessy and David A. Patterson. 2011. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann.

[16] T.A. Henzinger, C.M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova. 2013. Quantitative Relaxation of Concurrent Data Structures. In *Proc. Symposium on Principles of Programming Languages (POPL)*. ACM.

[17] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. 2000. *Computer-Aided Reasoning: An Approach*. Kluwer.

[18] Brian W. Kernighan and Dennis M. Ritchie. 2000. *The C Programming Language*. Prentice Hall.

[19] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proc. ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. ACM, 207–220. https://doi.org/10.1145/1629575.1629596

[20] Donald E. Knuth. 2011. *The Art of Computer Programming, Volumes 1–4*. Addison Wesley.

[21] Ramana Kumar. 2015. *Self-compilation and self-verification*. PhD thesis. University of Cambridge.

[22] Ole Lehrmann Madsen, Birger Mø-Pedersen, and Kristen Nygaard. 1993. *Object-oriented Programming in the BETA Programming Language*. ACM Press/Addison Wesley.

[23] J. Liedtke. 1996. Toward Real Microkernels. *Commun. ACM* 39, 9 (Sept. 1996), 70–77. https://doi.org/10.1145/234215.234473

[24] Noam Nisan and Shimon Schocken. 2005. *The Elements of Computing Systems: Building a Modern Computer from First Principles*. MIT Press.

[25] Martin Richards and Colin Whitby-Strevens. 2009. *BCPL: The Language and its Compiler*. Cambridge University Press.

[26] Patty M. Sailer, Philip M. Sailer, and David R. Kaeli. 1996. *The DLX Instruction Set Architecture Handbook* (1st ed.). Morgan Kaufmann.

[27] Michael Sipser. 1996. *Introduction to the Theory of Computation*. International Thomson Publishing.

[28] Gerald Jay Sussman and Hal Abelson. 1996. *Structure and Interpretation of Computer Programs*. MIT Press, Second Edition.

[29] Y. Vizel, G. Weissenbacher, and S. Malik. 2015. Boolean Satisfiability Solvers and Their Applications in Model Checking. *Proc. IEEE* 103, 11 (2015), 2021–2035.

[30] Niklaus Wirth. 1973. *Systematic Programming: An Introduction*. Prentice Hall.

[31] Niklaus Wirth. 1976. *Algorithms + Data Structures = Programs*. Prentice Hall.

[32] Niklaus Wirth. 1996. *Compiler Construction*. Addison Wesley.