How Language Informs Computer Science

The extensible, expandable, and expendable Ariane 5 is the workhorse of the European Space Agency. The humble spaceship was designed in the 90s to deliver payloads into low Earth orbit such as observational satellites, telecommunications equipment, and scientific modules for the International Space Station. Aeronautical engineers refer to the Ariane 5 as an expendable launch system. It is designed to fly into space, release its satellite payload into orbit or rendezvous with the ISS, and drifts off into space, never to be seen again. Since its inception, the Ariane 5 has proved to be a reliable and invaluable tool for the continued success of our endeavors in space, flying 66 missions to date and delivering such prestigious cargo as the Herschel Space Observatory, an infrared telescope that studies deep space, and the Rosetta space probe, which is on course to land on a comet that will pass through our solar system in 2014. The Ariane 5 is representative of the best space technology conceived by man; yet in spite of its impressive resume, the Ariane 5 will never deliver a human to space, even though its predecessor, the Ariane 4, was constructed for this very purpose. But why? Why deny such a powerful, reliable launch platform the highest honor we can bestow upon it: the task of bringing human beings into orbit? The fact is that the Ariane 5 began as a resounding disappointment.

The Ariane 5 stands 46-52 meters high and consists of two stages. *Photo by Philippe Semanaz/Wikipedia*

In June of 1996, thirty-seven seconds into her inaugural flight, Ariane 5 exploded and scattered debris over forty kilometers. Engineers determined after the incident that a fault in the

spacecraft's flight control software initiated the self-destruct sequence. It's preferable that the spaceship explode mid-flight and disintegrate into many small pieces, rather than crash back down to Earth whole. At the center of the guidance systems on launch vehicles like the Ariane 5 (and indeed virtually anything that flies, floats, or burns rubber) is a computer and a program that controls it. The Ariane 5 was brought down by an error in the code that the programmers oversaw when they were writing it. The actual bug, as we may aptly describe it, is a common occurrence in all software and is called an *overflow error*.

Now for a quick lecture on computer architecture. You probably know whether the computer you are using to read this article has a 32 bit or 64 bit processor. Ever wonder what this means? Your computer represents numbers as a sequence of bits–the ubiquitous 1s and 0s. The maximum length of this sequence is fixed: either 32 bits or 64 bits, depending on your processor's architecture. The more bits you have, the larger the number you can represent. But what happens if you try to represent a number larger than 32 or 64 bits? Say for example, that x and y are integers in memory on a 32 bit computer. Say z is the maximum number that can be represented by the CPU. If $x + y > z$, then attempting to store this value ($x = x + y$) results in an overflow error.

So what happens next? As it turns out, programmers are lazy people. The term for the behavior in general is *undefined*; we don't know what happens next and don't really care. Except that some people do care. The inventors of the Ada programming language, in which the guidance system of the Ariane 5 was written, cared a lot about this.

The Ariane 5 was brought down by exactly this kind of bug. The control system used a 16 bit integer as part of its mid-flight trajectory calculation. At some point, the software stored a

number too large to be represented by 16 bits – larger than $2^{16} = 65{,}536$ – and the guidance system believed for a few milliseconds that the rocket was actually pointed 90 degrees off of its planned trajectory. If an error like this occurs in Ada, the program will throw an *exception*, which kills the execution of the program. Luckily, the engineers of the Ariane 5 planned for this contingency: no control system, no rocket. The computer initiated the self-destruct sequence. This bug was an unforeseen consequence of transferring the guidance system software from the Ariane 4 to the Ariane 5. The control system worked fine on the older, less robust rocket, where the limitations of the computer hardware were of great concern. The Ariane 5's guidance computer was faster, more precise, and had more memory. In essence, it was able to compute with bigger numbers that don't fit into 16 bits; but since this number was only a small part of a very large piece of software, the engineers overlooked the problem.

“The answer is that the modern computer scientist must also be somewhat of a linguist.”

This is an inescapable challenge in all fields of computer science. Your web browser, word processor, Halo 4, even your Windows 15 or Mac OSX 17,003 are likely fraught with overflow errors, as well as a plethora of other bugs and glitches. Although they don't always bring down a rocket, computer bugs are a plague on the modern world. They crash your computer, cause you to lose your document, or make you type “Hel╝▀▒0x2ff1▄◊ Segmentation fault (core dumped)” when you mean “Hello, world!”. However, they also introduce vulnerabilities that threaten the security of the Internet and the financial institutions that rely on it. It seems society crucially depends on solving these problems. Besides being more careful, what are programmers doing to combat these sorts of catastrophic bugs? The answer is that the modern computer scientist must also be somewhat of a linguist.

The field of linguistics includes a wide range of topics relating to the form and evolution of language, including phonetics, morphology, syntax, and semantics to name a few. Phonetics and morphology concern the sounds of language as they evolve in tandem with society and culture. We can think of *syntax* as the way sentences are constructed in the language, or its grammar. For example, if we break the following sentence "*Readers of Wired have too much time on their hands"* into its grammatical components, we will find that each word has a function and a relationship to other words in the sentence. (*Readers* and *Wired* are both nouns, *of* is a preposition. *Wired* following *of* forms a prepositional phase modifying the noun *Readers. Readers of Wired* is the subject of the sentence.) Lastly, *semantics* is the study of the *meaning* of language. Syntax is very concrete, but the meaning of language is quite vague and at first seems only to have philosophical merit. What does the sentence about readers of *Wired* mean to you? Does it mean that readers of *Wired* should spend their time doing something else? How much time is too much time?

In recent years, there has been a huge surge in applying the linguist's methodology to the study of computation. After all, a programming language like Python, Java, or Ada is just that: a language. In fact, all programming languages have a syntax just like French, Spanish, Latin, or English; for without a syntax, computer wouldn't understand programs. As surprising as this may seem, the strangest fact is that computer scientists also study the semantics, or meaning, of programming languages.

As it turns out, computer programs are not only tools to accomplish specific tasks, but complex and intricate mathematical objects. In describing its semantics, computer scientists hope to establish the mathematical meaning of a programming language in order to understand how it

will behave under all circumstances. This methodology informs how the modern computer

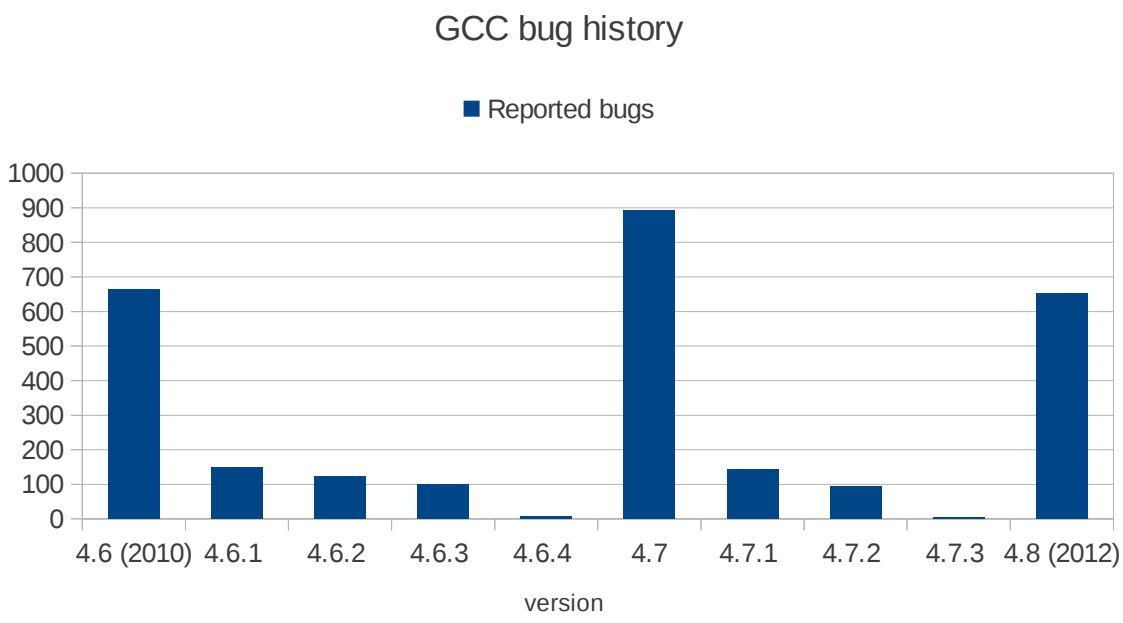
```
signature SEMANTICS = sig

    (* domains *)

    type Id = string;       (* variable name *)
    type Loc = int;    (* Location in memory *)
    datatype EV = Bool of bool | Int of int;          (* evaluable *)
    datatype SV = Stored of EV | Unused | Undefined;    (* storable *)
    datatype DV = Const of EV                         (* denotable *)
                | Var of Loc
                | Procedure of ((Id * DV) list * Id list * Block)
                | Continuation of Command list
                | Unbound;

    type Environment = (Id * DV) list;       (* program environment *)
    type Store = SV array;                              (* store *)

    (* semantic equations *)

    val meaning : Program -> Store;
    val perform : Environment * Store * Block -> Store;
    val execute : Environment * Store * Command list -> Store;
    val elaborate : Environment * Store * Declaration list -> Environment * Store;
    val evaluate : Environment * Store * Expr -> EV;
    val enumerate : numeral' -> int;

end

```

Real world semantics. *Image Credit: Christopher Patton*

scientist approaches the task of writing good code; he or she must now understand programs in a rigorous, mathematical sense. Why? Through a process known as program verification, researchers develop a formal semantics (in other words, a mathematical description of a particular programming language) and use it to determine if a program has an error in it, as well as the circumstances under which the bug would occur. This technique is used ubiquitously today for all sorts of programs. When Intel produces a new CPU, they put it through a vast array of tests to make sure that all combinations of instructions work properly. Indeed, program verification would have been the saving grace of the Ariane 5; a particular type of verification called axiomatic semantics is perfectly suited for the task of finding overflow errors in porgrams.

Program verification applies to conditions that we can often have a computer check

automatically. This means we can write a program to verify other programs. Sound strange? Sound impossible? In fact, researchers believe that there is a large set of questions about programs that can't be answered automatically in this way. Such problems still require a human touch. This has lead to the evolution of some imaginative new technologies. On the cutting edge is the DARPA Crowd-sourced Formal Verification (CSFV) Program. This military-funded research initiative is designed to prove that the control systems for critical applications like missiles, fighter jets, or spacecraft (like the Ariane 5) function properly. The idea behind CSFV is to develop a series of Internet-based games made available for the public to play. The games are essentially puzzles; but, through the magic of semantics, a solution to a puzzle corresponds to mathematical proof that a function of a particular weapons systems works. Patriotism at it's geekiest.



Typical frequency of software bugs for a large project over time. GCC is a popular programming tool with major releases about every year. Major releases introduce new features, and often, new bugs. Subsequent minor releases are for addressing these problems. *Source: http://gcc.gnu.org/bugzilla/*

Crowd-sourcing is just one of the many exciting methodologies coming out of the field of programming language semantics. The potential of program verification to solve critical problems now and for future generations is hard to overstate, but its impact on everyday life is possibly the most crucial result. Faster and more reliable computers are valuable to all of us. It seems that we'll be seeing a lot more programs like CSFV.

Christopher Patton for *Wired*