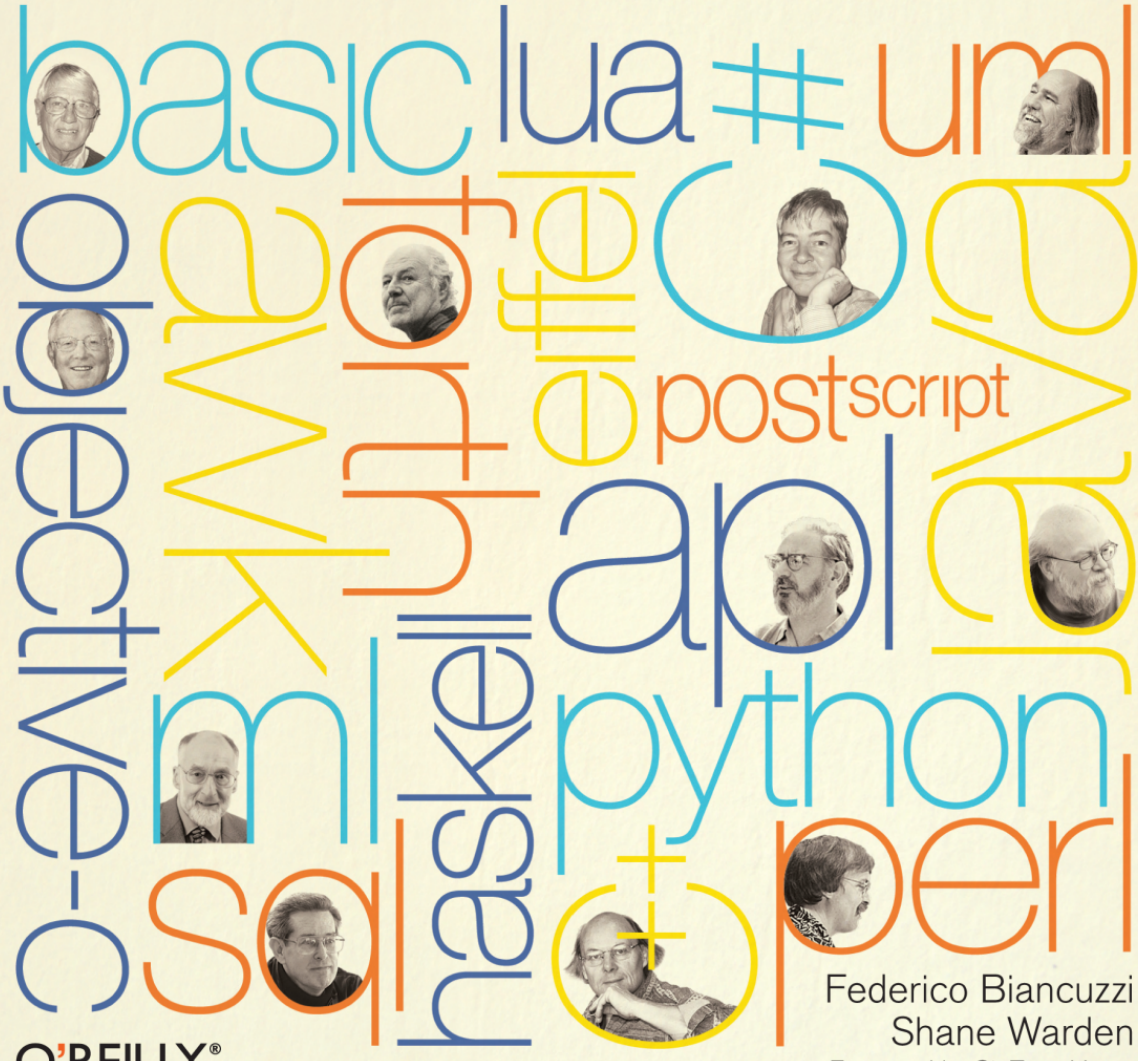
/THEORY/IN/PRACTICE

Masterminds of Programming

Conversations with the Creators of Major Programming Languages



Federico Biancuzzi
Shane Warden

Foreword by Sir Tony Hoare

O'REILLY®

Masterminds of Programming

Edited by Federico Biancuzzi and Shane Warden

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Masterminds of Programming
Edited by Federico Biancuzzi and Shane Warden

Published by O'Reilly Media, Inc. 1005 Gravenstein Highway North, Sebastopol, CA 95472

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Andy Oram
Production Editor: Rachel Monaghan
Indexer: Angela Howard
Proofreader: Nancy Kotary
Cover Designer: Monica Kamsvaag
Interior Designer: Marcia Friedman

Printing History:

March 2009: First Edition.

ISBN: 978-0-596-51517-1
[LSI] [2014-03-21]

CONTENTS

Foreword

Programming language design is a fascinating topic. There are so many programmers who think they can design a programming language better than one they are currently using; and there are so many researchers who believe they can design a programming language better than any that are in current use. Their beliefs are often justified, but few of their designs ever leave the designer's bottom drawer. You will not find them represented in this book.

Programming language design is a serious business. Small errors in a language design can be conducive to large errors in an actual program written in the language, and even small errors in programs can have large and extremely costly consequences. The vulnerabilities of widely used software have repeatedly allowed attack by malware to cause billions of dollars of damage to the world economy. The safety and security of programming languages is a recurrent theme of this book.

Programming language design is an unpredictable adventure. Languages designed for universal application, even when supported and sponsored by vast organisations, end up sometimes in just a niche market. In contrast, languages designed for limited or local use can win a broad clientele, sometimes in environments and for applications that their designers never dreamed of. This book concentrates on languages of the latter kind.

These successful languages share a significant characteristic: each of them is the brainchild of a single person or a small team of like-minded enthusiasts. Their designers are masterminds of programming; they have the experience, the vision, the energy, the persistence, and the sheer genius to drive the language through its initial implementation, through its evolution in the light of experience, and through its standardisation by usage (de facto) and by committee (de jure).

In this book the reader will meet this collection of masterminds in person. Each of them has granted an extended interview, telling the story of his language and the factors that lie behind its success. The combined role of good decisions and good luck is frankly acknowledged. And finally, the publication of the actual words spoken in the interview gives an insight into the personality and motivations of the designer, which is as fascinating as the language design itself.

—Sir Tony Hoare

Sir Tony Hoare, winner of an ACM Turing Award and a Kyoto Award, has been a leader in research into computing algorithms and programming languages for 50 years. His first academic paper, written in 1969, explored the idea of proving the correctness of programs, and suggested that a goal of programming language design was to make it easier to write correct programs. He is delighted to see the idea spread gradually among programming language designers.

Preface

Writing software is hard—at least, writing software that stands up under tests, time, and different environments is hard. Not only has the software engineering field struggled to make writing software easier over the past five decades, but languages have been designed to make it easier. But what makes it hard in the first place?

Most of the books and the papers that claim to address this problem talk about architecture, requirements, and similar topics that focus on the *software*. What if the hard part was in the *writing*? To put it another way, what if we saw our jobs as programmers more in terms of communication—*language*—and less in terms of engineering?

Children learn to talk in their first years of life, and we start teaching them how to read and write when they are five or six years old. I don't know any great writer who learned to read and write as an adult. Do you know any great programmer who learned to program late in life?

And if children can learn foreign languages much more easily than adults, what does this tell us about learning to program—an activity involving a new language?

Imagine that you are studying a foreign language and you don't know the name of an object. You can describe it with the words that you know, hoping someone will understand what you mean. Isn't this what we do every day with software? We describe the object we have in our mind with a programming language, hoping the description will be clear enough to the compiler or interpreter. If something doesn't work, we bring up the picture again in our mind and try to understand what we missed or misdescribed.

With these questions in mind, I chose to launch a series of investigations into why a programming language is created, how it's technically developed, how it's taught and learned, and how it evolves over time.

Shane and I had the great privilege to let 27 great designers guide us through our journey, so that we have been able to collect their wisdom and experience for you.

In *Masterminds of Programming*, you will discover some of the thinking and steps needed to build a successful language, what makes it popular, and how to approach the current problems that its programmers are facing. So if you want to learn more about successful programming language design, this book surely can help you.

If you are looking for inspiring thoughts regarding software and programming languages, you will need a highlighter, or maybe two, because I promise that you will find plenty of them throughout these pages.

—Federico Biancuzzi

Organization of the Material

The chapters in this book are ordered to provide a varied and provocative perspective as you travel through it. Savor the interviews and return often.

Chapter 1, *C++*, interviews Bjarne Stroustrup.

Chapter 2, *Python*, interviews Guido van Rossum.

Chapter 3, *APL*, interviews Adin D. Falkoff.

Chapter 4, *Forth*, interviews Charles H. Moore.

Chapter 5, *BASIC*, interviews Thomas E. Kurtz.

Chapter 6, *AWK*, interviews Alfred Aho, Peter Weinberger, and Brian Kernighan.

Chapter 7, *Lua*, interviews Luiz Henrique de Figueiredo and Roberto Ierusalimschy.

Chapter 8, *Haskell*, interviews Simon Peyton Jones, Paul Hudak, Philip Wadler, and John Hughes.

Chapter 9, *ML*, interviews Robin Milner.

Chapter 10, *SQL*, interviews Don Chamberlin.

Chapter 11, *Objective-C*, interviews Tom Love and Brad Cox.

Chapter 12, *Java*, interviews James Gosling.

Chapter 13, *C#*, interviews Anders Hejlsberg.

Chapter 14, *UML*, interviews Ivar Jacobson, James Rumbaugh, and Grady Booch.

Chapter 15, *Perl*, interviews Larry Wall.

Chapter 16, *PostScript*, interviews Charles Geschke and John Warnock.

Chapter 17, *Eiffel*, interviews Bertrand Meyer.

Contributors lists the biographies of all the contributors.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, filenames, and utilities.

`Constant width`
: Indicates the contents of computer files and generally anything found in programs.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/9780596515171

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our website at:

http://www.oreilly.com

Safari® Books Online

When you see a Safari® Books Online icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at *http://my.safaribooksonline.com*.

CHAPTER ONE

C++

C++ occupies an interesting space among languages: it is built on the foundation of C, incorporating object-orientation ideas from Simula; standardized by ISO; and designed with the mantras "you don't pay for what you don't use" and "support user-defined and built-in types equally well." Although popularized in the 80s and 90s for OO and GUI programming, one of its greatest contributions to software is its pervasive generic programming techniques, exemplified in its Standard Template Library. Newer languages such as Java and C# have attempted to replace C++, but an upcoming revision of the C++ standard adds new and long-awaited features. Bjarne Stroustrup is the creator of the language and still one of its strongest advocates.

Design Decisions

Why did you choose to extend an existing language instead of creating a new one?

Bjarne Stroustrup: When I started—in 1979—my purpose was to help programmers build systems. It still is. To provide genuine help in solving a problem, rather than being just an academic exercise, a language must be complete for the application domain. That is, a non-research language exists to solve a problem. The problems I was addressing related to operating system design, networking, and simulation. I—and my colleagues—needed a language that could express program organization as could be done in Simula (that's what people tend to call object-oriented programming), but also write efficient low-level code, as could be done in C. No language that could do both existed in 1979, or I would have used it. I didn't particularly want to design a new programming language; I just wanted to help solve a few problems.

Given that, building on an existing language makes a lot of sense. From the base language, you get a basic syntactic and semantic structure, you get useful libraries, and you become part of a culture. Had I not built on C, I would have based C++ on some other language. Why C? I had Dennis Ritchie, Brian Kernighan, and other Unix greats just down (or across) the hall from me in Bell Labs' Computer Science Research Center, so the question may seem redundant. But it was a question I took seriously.

In particular, C's type system was informal and weakly enforced (as Dennis Ritchie said, "C is a strongly typed, weakly checked language"). The "weakly checked" part worried me and causes problems for C++ programmers to this day. Also, C wasn't the widely used language it is today. Basing C++ on C was an expression of faith in the model of computation that underlies C (the "strongly typed" part) and an expression of trust in my colleagues. The choice was made based on knowledge of most higher-level programming languages used for systems programming at the time (both as a user and as an implementer). It is worth remembering that this was a time when most work "close to the hardware" and requiring serious performance was still done in assembler. Unix was a major breakthrough in many ways, including its use of C for even the most demanding systems programming tasks.

So, I chose C's basic model of the machine over better-checked type systems. What I really wanted as the framework for programs was Simula's classes, so I mapped those into the C model of memory and computation. The result was something that was extremely expressive and flexible, yet ran at a speed that challenged assembler without a massive runtime support system.

Why did you choose to support multiple paradigms?

Bjarne: Because a combination of programming styles often leads to the best code, where "best" means code that most directly expresses the design, runs faster, is most maintainable, etc. When people challenge that statement, they usually do so by either defining their favorite programming style to include every useful construct (e.g., "generic programming is simply a form of OO") or excluding application areas (e.g., "everybody has a 1GHz, 1GB machine").

Java focuses solely on object-oriented programming. Does this make Java code more complex in some cases where C++ can instead take advantage of generic programming?

Bjarne: Well, the Java designers—and probably the Java marketers even more so—emphasized OO to the point where it became absurd. When Java first appeared, claiming purity and simplicity, I predicted that if it succeeded Java would grow significantly in size and complexity. It did.

For example, using casts to convert from `Object` when getting a value out of a container (e.g., `(Apple)c.get(i)`) is an absurd consequence of not being able to state what type the objects in the container is supposed have. It's verbose and inefficient. Now Java has generics, so it's just a bit slow. Other examples of increased language complexity (helping the programmer) are enumerations, reflection, and inner classes.

The simple fact is that complexity will emerge somewhere, if not in the language definition, then in thousands of applications and libraries. Similarly, Java's obsession with putting every algorithm (operation) into a class leads to absurdities like classes with no data consisting exclusively of static functions. There are reasons why math uses `f(x)` and `f(x,y)` rather than `x.f()`, `x.f(y)`, and `(x,y).f()`—the latter is an attempt to express the idea of a "truly object-oriented method" of two arguments and to avoid the inherent asymmetry of `x.f(y)`.

C++ addresses many of the logical as well as the notational problems with object orientation through a combination of data abstraction and generic programming techniques. A classical example is `vector<T>` where `T` can be any type that can be copied—including built-in types, pointers to OO hierarchies, and user-defined types, such as strings and complex numbers. This is all done without adding runtime overheads, placing restrictions on data layouts, or having special rules for standard library components. Another example that does not fit the classical single-dispatch hierarchy model of OO is an operation that requires access to two classes, such as `operator*(Matrix,Vector)`, which is not naturally a "method" of either class.

One fundamental difference between C++ and Java is the way pointers are implemented. In some ways, you could say that Java doesn't have real pointers. What differences are there between the two approaches?

Bjarne: Well, of course Java has pointers. In fact, just about everything in Java is implicitly a pointer. They just call them *references*. There are advantages to having pointers implicit as well as disadvantages. Separately, there are advantages to having true local objects (as in C++) as well as disadvantages.

C++'s choice to support stack-allocated local variables and true member variables of every type gives nice uniform semantics, supports the notion of value semantics well, gives compact layout and minimal access costs, and is the basis for C++'s support for general resource management. That's major, and Java's pervasive and implicit use of pointers (aka references) closes the door to all that.

Consider the layout tradeoff: in C++ a `vector<complex>(10)` is represented as a handle to an array of 10 complex numbers on the free store. In all, that's 25 words: 3 words for the vector, plus 20 words for the complex numbers, plus a 2-word header for the array on the free store (heap). The equivalent in Java (for a user-defined container of objects of user-defined types) would be 56 words: 1 for the reference to the container, plus 3 for the container, plus 10 for the references to the objects, plus 20 for the objects, plus 24 for the free store headers for the 12 independently allocated objects. Obviously, these numbers are approximate because the free store (heap) overhead is implementation defined in both languages. However, the conclusion is clear: by making references ubiquitous and implicit, Java may have simplified the programming model and the garbage collector implementation, but it has increased the memory overhead dramatically—and increased the memory access cost (requiring more indirect accesses) and allocation overheads proportionally.

What Java doesn't have—and good for Java for that—is C and C++'s ability to misuse pointers through pointer arithmetic. Well-written C++ doesn't suffer from that problem either: people use higher-level abstractions, such as iostreams, containers, and algorithms, rather than fiddling with pointers. Essentially all arrays and most pointers belong deep in implementations that most programmers don't have to see. Unfortunately, there is also lots of poorly written and unnecessarily low-level C++ around.

There is, however, an important place where pointers—and pointer manipulation—is a boon: the direct and efficient expression of data structures. Java's references are lacking here; for example, you can't express a `swap` operation in Java. Another example is simply the use of pointers for low-level direct access to (real) memory; for every system, some language has to do that, and often that language is C++.

The "dark side" of having pointers (and C-style arrays) is of course the potential for misuse: buffer overruns, pointers into deleted memory, uninitialized pointers, etc. However, in well-written C++ that is not a major problem. You simply don't get those problems with pointers and arrays used within abstractions (such as `vector`, `string`, `map`, etc.). Scoped resource management takes care of most needs; smart pointers and specialized handles can be used to deal with most of the rest. People whose experience is primarily C or old-style C++ find this hard to believe, but scope-based resource management is an immensely powerful tool and user-defined with suitable operations can address classical problems with less code than the old insecure hacks. For example, this is the simplest form of the classical buffer overrun and security problem:

```
char buf[MAX_BUF];
gets(buf); // Yuck!
```

Use a standard library string and the problem goes away:

```
string s;
cin >> s;    // read whitespace separated characters
```

These are obviously trivial examples, but suitable "strings" and "containers" can be crafted to meet essentially all needs, and the standard library provides a good set to start with.

What do you mean by "value semantics" and "general resource management"?

Bjarne: "Value semantics" is commonly used to refer to classes where the objects have the property that when you copy one, you get two independent copies (with the same value). For example:

```
X x1 = a;
X x2 = x1; // now x1==x2
x1 = b;    // changes x1 but not x2
           // now x1!=x2 ( provided X(a)!=X(b) )
```

This is of course what we have for usual numeric types, such as ints, doubles, complex numbers, and mathematical abstractions, such as vectors. This is a most useful notion, which C++ supports for built-in types and for any user-defined type for which we want it. This contrast to Java where built-in types such and `char` and `int` follow it, but user-defined types do not, and indeed cannot. As in Simula, all user-defined types in Java have reference semantics. In C++, a programmer can support either, as the desired semantics of a type requires. C# (incompletely) follows C++ in supporting user-defined types with value semantics.

"General resource management" refers to the popular technique of having a resource (e.g., a file handle or a lock) owned by an object. If that object is a scoped variable, the lifetime of the variable puts a maximum limit on the time the resource is held. Typically, a constructor acquires the resource and the destructor releases it. This is often called RAII (Resource Acquisition Is Initialization) and integrates beautifully with error handling using exceptions. Obviously, not every resource can be handled in this way, but many can, and for those, resource management becomes implicit and efficient.

"Close to the hardware" seems to be a guiding principle in designing C++. Is it fair to say that C++ was designed more bottom-up than many languages, which are designed top-down, in the sense that they try to provide abstractly rational constructs and force the compiler to fit these constructs to the available computing environment?

Bjarne: I think top-down and bottom-up are the wrong way to characterize those design decisions. In the context of C++ and other languages, "close to the hardware" means that the model of computation is that of the computer—sequences of objects in memory and operations as defined on objects of fixed size—rather than some mathematical abstraction. This is true for both C++ and Java, but not for functional languages. C++ differs from Java in that its underlying machine is the real machine rather than a single abstract machine.

The real problem is how to get from the human conception of problems and solutions to the machine's limited world. You can "ignore" the human concerns and end up with machine code (or the glorified machine code that is bad C code). You can ignore the machine and come up with a beautiful abstraction that can do anything at extraordinary cost and/or lack of intellectual rigor. C++ is an attempt to give a very direct access to hardware when you need it (e.g., pointers and arrays) while providing extensive abstraction mechanisms to allow high-level ideas to be expressed (e.g., class hierarchies and templates).

That said, there has been a consistent concern for runtime and space performance throughout the development of C++ and its libraries. This pervades both the basic language facilities and the abstraction facilities in ways that are not shared by all languages.

Using the Language

How do you debug? Do you have any suggestion for C++ developers?

Bjarne: By introspection. I study the program for so long and poke at it more or less systematically for so long that I have sufficient understanding to provide an educated guess where the bug is.

Testing is something else, and so is design to minimize errors. I intensely dislike debugging and will go a long way to avoid it. If I am the designer of a piece of software, I build it around interfaces and invariants so that it is hard to get seriously bad code to compile and run incorrectly. Then, I try hard to make it testable. Testing is the systematic search for errors. It is hard to systematically test badly structured systems, so I again recommend a clean structure of the code. Testing can be automated and is repeatable in a way that debugging is not. Having flocks of pigeons randomly peck at the screen to see if they can break a GUI-based application is no way to ensure quality systems.

Advice? It is hard to give general advice because the best techniques often depend on what is feasible for a given system in a given development environment. However: identify key interfaces that can be systematically tested and write test scripts that exercise those. Automate as much as you can and run those automated tests often. And do keep regression tests and run them frequently. Make sure that every entry point into the system and every output can be systematically tested. Compose your system out of quality components: monolithic programs are unnecessarily hard to understand and test.

At what level is it necessary to improve the security of software?

Bjarne: First of all: security is a systems issue. No localized or partial remedy will by itself succeed. Remember, even if all of your code was perfect, I could probably still gain access to your stored secrets if I could steal your computer or the storage device holding your backup. Secondly, security is a cost/benefit game: perfect security is probably beyond the reach for most of us, but I can probably protect my system sufficiently that "bad guys" will consider their time better spent trying to break into someone else's system. Actually, I prefer not to keep important secrets online and leave serious security to the experts.

But what about programming languages and programming techniques? There is a dangerous tendency to assume that every line of code has to be "secure" (whatever that means), even assuming that someone with bad intentions messes with some other part of the system. This is a most dangerous notion that leaves the code littered with unsystematic tests guarding against ill-formulated imagined threats. It also makes code ugly, large, and slow. "Ugly" leaves places for bugs to hide, "large" ensures incomplete testing, and "slow" encourages the use of shortcuts and dirty tricks that are among the most fertile sources of security holes.

I think the only permanent solution to security problems is in a simple security model applied systematically by quality hardware and/or software to selected interfaces. There has to be a place behind a barrier where code can be written simply, elegantly, and efficiently without worrying about random pieces of code abusing random pieces of other code. Only then can we focus on correctness, quality, and serious performance. The idea that anyone can provide an untrusted callback, plug-in, overrider, whatever, is plain silly. We have to distinguish between code that defends against fraud, and code that simply is protected against accidents.

I do not think that you can design a programming language that is completely secure and also useful for real-world systems. Obviously, that depends on the meaning of "secure" and "system." You could possibly achieve security in a domain-specific language, but my main domain of interest is systems programming (in a very broad meaning of that term), including embedded systems programming. I do think that type safety can and will be improved over what is offered by C++, but that is only part of the problem: type safety does not equal security. People who write C++ using lots of unencapsulated arrays, casts, and unstructured new and delete operations are asking for trouble. They are stuck in an 80s style of programming. To use C++ well, you have to adopt a style that minimizes type safety violations and manage resources (including memory) in a simple and systematic way.

Would you recommend C++ for some systems where practitioners are reluctant to use it, such as system software and embedded applications?

Bjarne: Certainly, I do recommend it and not everybody is reluctant. In fact, I don't see much reluctance in those areas beyond the natural reluctance to try something new in established organizations. Rather, I see steady and significant growth in C++ use. For example, I helped write the coding guidelines for the mission-critical software for Lockheed Martin's Joint Strike Fighter. That's an "all C++ plane." You may not be particularly keen on military planes, but there is nothing particularly military about the way C++ is used and well over 100,000 copies of the JSF++ coding rules have been downloaded from my home pages in less than a year, mostly by nonmilitary embedded systems developers, as far as I can tell.

C++ has been used for embedded systems since 1984, many useful gadgets have been programmed in C++, and its use appears to be rapidly increasing. Examples are mobile phones using Symbian or Motorola, the iPods, and GPS systems. I particularly like the use of C++ on the Mars rovers: the scene analysis and autonomous driving subsystems, much of the earth-based communication systems, and the image processing.

People who are convinced that C is necessarily more efficient than C++ might like to have a look at my paper entitled "Learning Standard C++ as a New Language" [*C/C++ Users Journal*, May 1999], which describes a bit of design philosophy and shows the result of a few simple experiments. Also, the ISO C++ standards committee issued a technical report on performance that addresses a lot of issues and myths relating to the use of C++ where performance matters (you can find it online searching for "Technical Report on C++ Performance").* In particular, that report addresses embedded systems issues.

* *http://www.open-std.org/JTC1/sc22/wg21/docs/TR18015.pdf*

Kernels like Linux's or BSD's are still written in C. Why haven't they moved to C++? Is it something in the OO paradigm?

Bjarne: It's mostly conservatism and inertia. In addition, GCC was slow to mature. Some people in the C community seem to maintain an almost willful ignorance based on decade-old experiences. Other operating systems and much systems programming and even hard real-time and safety-critical code has been written in C++ for decades. Consider some examples: Symbian, IBM's OS/400 and K42, BeOS, and parts of Windows. In general, there is a lot of open source C++ (e.g., KDE).

You seem to equate C++ use with OO. C++ is not and was never meant to be just an object-oriented programming language. I wrote a paper entitled "Why C++ is not just an Object-Oriented Programming Language" in 1995; it is available online.* The idea was and is to support multiple programming styles ("paradigms," if you feel like using long words) and their combinations. The most relevant other paradigm in the context of high-performance and close-to-the-hardware use is generic programming (sometimes abbreviated to GP). The ISO C++ standard library is itself more heavily GP than OO through its framework for algorithms and containers (the STL). Generic programming in the typical C++ style relying heavily on templates is widely used where you need both abstraction and performance.

I have never seen a program that could be written better in C than in C++. I don't think such a program could exist. If nothing else, you can write C++ in a style close to that of C. There is nothing that requires you to go hog-wild with exceptions, class hierarchies, or templates. A good programmer uses the more advanced features where they help more directly to express ideas and do so without avoidable overheads.

Why should a programmer move his code from C to C++? What advantages would he have using C++ as a generic programming language?

Bjarne: You seem to assume that code first was written in C and that the programmer started out as a C programmer. For many—probably most—C++ programs and C++ programmers, that has not been the case for quite a while. Unfortunately, the "C first" approach lingers in many curricula, but it is no longer something to take for granted.

Someone might switch from C to C++ because they found C++'s support for the styles of programming usually done with C is better than C's. The C++ type checking is stricter (you can't forget to declare a function or its argument types) and there is type-safe notational support for many common operations, such as object creation (including initialization) and constants. I have seen people do that and be very happy with the problems they left behind. Usually, that's done in combination with the adoption of some C++ libraries that may or may not be considered object-oriented, such as the standard vector, a GUI library, or some application-specific library.

* *http://www.research.att.com/~bs/oopsla.pdf*

Just using a simple user-defined type, such as `vector`, `string`, or `complex`, does not require a paradigm shift. People can—if they so choose—use those just like the built-in types. Is someone using `std::vector` "using OO"? I would say no. Is someone using a C++ GUI without actually adding new functionality "using OO"? I'm inclined to say yes, because their use typically requires the users to understand and use inheritance.

Using C++ as "a generic-programming programming language" gives you the standard containers and algorithms right out of box (as part of the standard library). That is major leverage in many applications and a major step up in abstraction from C. Beyond that, people can start to benefit from libraries, such as Boost, and start to appreciate some of the functional programming techniques inherent in generic programming.

However, I think the question is slightly misleading. I don't want to represent C++ as "an OO language" or "a GP language"; rather, it is a language supporting:

- C-style programming
- Data abstraction
- Object-oriented programming
- Generic programming

Crucially, it supports programming styles that combines those ("multiparadigm programming" if you must) and does so with a bias toward systems programming.

OOP and Concurrency

The average complexity and size (in number of lines of code) of software seems to grow year after year. Does OOP scale well to this situation or just make things more complicated? I have the feeling that the desire to make reusable objects makes things more complicated and, in the end, it doubles the workload. First, you have to design a reusable tool. Later, when you need to make a change, you have to write something that exactly fits the gap left by the old part, and this means restrictions on the solution.

Bjarne: That's a good description of a serious problem. OO is a powerful set of techniques that can help, but to be a help, it must be used well and for problems where the techniques have something to offer. A rather serious problem for all code relying on inheritance with statically checked interfaces is that to design a good base class (an interface to many, yet unknown, classes) we require a lot of foresight and experience. How does the designer of the base class (abstract class, interface, whatever you choose to call it) know that it specifies all that is needed for all classes that will be derived from it in the future? How does the designer know that what is specified can be implemented reasonably by all classes that will be derived from it in the future? How does the designer of the base class know that what is specified will not seriously interfere with something that is needed by some classes that will be derived from it in the future?

In general, we can't know that. In an environment where we can enforce our design, people will adapt—often by writing ugly workarounds. Where no one organization is in charge, many incompatible interfaces emerge for essentially the same functionality.

Nothing can solve these problems in general, but generic programming seems to be an answer in many important cases where the OO approach fails. A noteworthy example is simply containers: we cannot express the notion of being an element well through an inheritance hierarchy, and we can't express the notion of being a container well through an inheritance hierarchy. We can, however, provide effective solutions using generic programming. The STL (as found in the C++ standard library) is an example.

Is this problem specific to C++, or does it afflict other programming languages as well?

Bjarne: The problem is common to all languages that rely on statically checked interfaces to class hierarchies. Examples are C++, Java, and C#, but not dynamically typed languages, such as Smalltalk and Python. C++ addresses that problem through generic programming, where the C++ containers and algorithms in standard library provide a good example. The key language feature here is templates, providing a late type-checking model that gives a compile time equivalent to what the dynamically typed languages do at runtime. Java's and C#'s recent addition of "generics" are attempts to follow C++'s lead here, and are often—incorrectly, I think—claimed to improve upon templates.

"Refactoring" is especially popular as an attempt to address that problem by the brute force technique of simply reorganizing the code when it has outlived its initial interface design.

If this is a problem of OO in general, how can we be sure that the advantages of OO are more valuable than the disadvantages? Maybe the problem that a good OO design is difficult to achieve is the root of all other problems.

Bjarne: The fact that there is a problem in some or even many cases doesn't change the fact that many beautiful, efficient, and maintainable systems have been written in such languages. Object-oriented design is one of the fundamental ways of designing systems and statically checked interfaces provide advantages as well as this problem.

There is no one "root of all evil" in software development. Design is hard in many ways. People tend to underestimate the intellectual and practical difficulties involved in building a significant system involving software. It is not and will not be reduced to a simple mechanical "assembly line" process. Creativity, engineering principles, and evolutionary change are needed to create a satisfactory large system.

Are there links between the OO paradigm and concurrency? Does the current pervasive need for improved concurrency change the implementation of designs or the nature of OO designs?

Bjarne: There is a very old link between object-oriented programming and concurrency. Simula 67, the programming language that first directly supported object-oriented programming, also provided a mechanism for expressing concurrent activities.

The first C++ library was a library supporting what today we would call *threads*. At Bell Labs, we ran C++ on a six-processor machine in 1988 and we were not alone in such uses. In the 90s there were at least a couple of dozen experimental C++ dialects and libraries attacking problems related to distributed and parallel programming. The current excitement about multicores isn't my first encounter with concurrency. In fact, distributed computing was my Ph.D. topic and I have followed that field ever since.

However, people who first consider concurrency, multicores, etc., often confuse themselves by simply underestimating the cost of running an activity on a different processor. The cost of starting an activity on another processor (core) and for that activity to access data in the "calling processor's" memory (either copying or accessing "remotely") can be 1,000 times (or more) higher than we are used to for a function call. Also, the error possibilities are significantly different as soon as you introduce concurrency. To effectively exploit the concurrency offered by the hardware, we need to rethink the organization of our software.

Fortunately, but confusingly, we have decades' worth of research to help us. Basically, there is so much research that it's just about impossible to determine what's real, let alone what's best. A good place to start looking would be the HOPL-III paper about Emerald. That language was the first to explore the interaction between language issues and systems issues, taking cost into account. It is also important to distinguish between data parallel programming as has been done for decades—mostly in FORTRAN—for scientific calculations, and the use of communicating units of "ordinary sequential code" (e.g., processes and threads) on many processors. I think that for broad acceptance in this brave new world of many "cores" and clusters, a programming system must support both kinds of concurrency, and probably several varieties of each. This is not at all easy, and the issues go well beyond traditional programming language issues—we will end up looking at language, systems, and applications issues in combination.

Is C++ ready for concurrency? Obviously we can create libraries to handle everything, but does the language and standard library need a serious review with concurrency in mind?

Bjarne: Almost. C++0x will be. To be ready for concurrency, a language first has to have a precisely specified memory model to allow compiler writers to take advantage of modern hardware (with deep pipelines, large caches, branch-prediction buffers, static and dynamic instruction reordering, etc.). Then, we need a few small language extensions: thread-local storage and atomic data types. Then, we can add support for concurrency as libraries. Naturally, the first new standard library will be a threads library allowing portable programming across systems such as Linux and Windows. We have of course had such libraries for many years, but not standard ones.

Threads plus some form of locking to avoid data races is just about the worst way to directly exploit concurrency, but C++ needs that to support existing applications and to maintain its role as a systems programming language on traditional operating systems. Prototypes of this library exist—based on many years of active use.

One key issue for concurrency is how you "package up" a task to be executed concurrently with other tasks. In C++, I suspect the answer will be "as a function object." The object can contain whatever data is needed and be passed around as needed. C++98 handles that well for named operations (named classes from which we instantiate function objects), and the technique is ubiquitous for parameterization in generic libraries (e.g., the STL). C++0x makes it easier to write simple "one-off" function objects by providing "lambda functions" that can be written in expression contexts (e.g., as function arguments) and generates function objects ("closures") appropriately.

The next steps are more interesting. Immediately post-C++0x, the committee plans for a technical report on libraries. This will almost certainly provide for thread pools and some form of work stealing. That is, there will be a standard mechanism for a user to request relatively small units of work ("tasks") to be done concurrently without fiddling with thread creation, cancellation, locking, etc., probably built with function objects as tasks. Also, there will be facilities for communicating between geographically remote processes through sockets, iostreams, and so on, rather like `boost::networking`.

In my opinion, much of what is interesting about concurrency will appear as multiple libraries supporting logically distinct concurrency models.

Many modern systems are componentized and spread out over a network; the age of web applications and mashups may accentuate that trend. Should a language reflect those aspects of the network?

Bjarne: There are many forms of concurrency. Some are aimed at improving the throughput or response time of a program on a single computer or cluster, some are aimed at dealing with geographical distribution, and some are below the level usually considered by programmers (pipelining, caching, etc.).

C++0*x* will provide a set of facilities and guarantees that saves programmers from the lowest-level details by providing a "contract" between machine architects and compiler writers—a "machine model." It will also provide a threads library providing a basic mapping of code to processors. On this basis, other models can be provided by libraries. I would have liked to see some simpler-to-use, higher-level concurrency models supported in the C++0*x* standard library, but that now appears unlikely. Later—hopefully, soon after C++0*x*—we will get more libraries specified in a technical report: thread pools and futures, and a library for I/O streams over wide area networks (e.g., TCP/IP). These libraries exist, but not everyone considers them well enough specified for the standard.

Years ago, I hoped that C++0*x* would address some of C++'s long-standing problems with distribution by specifying a standard form of marshalling (or serialization), but that didn't happen. So, the C++ community will have to keep addressing the higher levels of distributed computing and distributed application building through nonstandard libraries and/or frameworks (e.g., CORBA or .NET).

The very first C++ library (really the very first C with classes) library, provided a lightweight form of concurrency and over the years, hundreds of libraries and frameworks for

concurrent, parallel, and distributed computing have been built in C++, but the community has not been able to agree on standards. I suspect that part of the problem is that it takes a lot of money to do something major in this field, and that the big players preferred to spend their money on their own proprietary libraries, frameworks, and languages. That has not been good for the C++ community as a whole.

Future

Will we ever see C++ 2.0?

Bjarne: That depends on what you mean by "C++ 2.0." If you mean a new language built more or less from scratch providing all of the best of C++ but none of what's bad (for some definitions of "good" and "bad"), the answer is "I don't know." I would like to see a major new language in the C++ tradition, but I don't see one on the horizon, so let me concentrate on the next ISO C++ standard, nicknamed C++0x.

It will be a "C++ 2.0" to many, because it will supply new language features and new standard libraries, but it will be almost 100% compatible with C++98. We call it C++0*x*, hoping that it'll become C++09. If we are slow—so that that *x* has to become hexadecimal—I (and others) will be quite sad and embarrassed.

C++0x will be almost 100% compatible with C++98. We have no particular desire to break your code. The most significant incompatibilities come from the use of a few new keywords, such as `static_assert`, `constexpr`, and `concept`. We have tried to minimize impact by choosing new keywords that are not heavily used. The major improvements are:

- Support for modern machine architectures and concurrency: a machine model, a thread library, thread local storage and atomic operations, and an asynchronous value return mechanism ("futures").
- Better support for generic programming: concepts (a type system for types, combinations of types, and combinations of types and integers) to give better checking of template definitions and uses, and better overloading of templates. Type deduction based on initializers (`auto`), generalized initializer lists, generalized constant expressions (`constexpr`), lambda expressions, and more.
- Many "minor" language extensions, such as static assertions, move semantics, improved enumerations, a name for the null pointer (`nullptr`), etc.
- New standard libraries for regular expression matching, hash tables (e.g., `unordered_map`), "smart" pointers, etc.

For complete details, see the website of the "C++ Standards Committee."* For an overview, see my online C++0x FAQ.†

* *http://www.open-std.org/jtc1/sc22/wg21/*

† *http://www.research.att.com/~bs/C++0xFAQ.html*

Please note that when I talk about "not breaking code," I am referring to the core language and the standard library. Old code will of course be broken if it uses nonstandard extensions from some compiler provider or antique nonstandard libraries. In my experience, when people complain about "broken code" or "instability" they are referring to proprietary features and libraries. For example, if you change operating systems and didn't use one of the portable GUI libraries, you probably have some work to do on the user interface code.

What stops you from creating a major new language?

Bjarne: Some key questions soon emerge:

- What problem would the new language solve?
- Who would it solve problems for?
- What dramatically new could be provided (compared to every existing language)?
- Could the new language be effectively deployed (in a world with many well-supported languages)?
- Would designing a new language simply be a pleasant distraction from the hard work of helping people build better real-world tools and systems?

So far, I have not been able to answer those questions to my satisfaction.

That doesn't mean that I think that C++ is the perfect language of its kind. It is not; I'm convinced that you could design a language about a tenth of the size of C++ (whichever way you measure size) providing roughly what C++ does. However, there has to be more to a new language that just doing what an existing language can, but slightly better and slightly more elegantly.

What do the lessons about the invention, further development, and adoption of your language say to people developing computer systems today and in the foreseeable future?

Bjarne: That's a big question: can we learn from history? If so, how? What kind of lessons can we learn? During the early development of C++, I articulated a set of "rules of thumb," which you can find in *The Design and Evolution of C++* [Addison-Wesley], and also discussed in my two HOPL papers. Clearly, any serious language design project needs a set of principles, and as soon as possible, these principles need to be articulated. That's actually a conclusion from the C++ experience: I didn't articulate C++'s design principles early enough and didn't get those principles understood widely enough. As a result, many people invented their own rationales for C++'s design; some of those were pretty amazing and led to much confusion. To this day, some see C++ as little more than a failed attempt to design something like Smalltalk (no, C++ was not supposed to be "like Smalltalk"; it follows the Simula model of OO), or as nothing but an attempt to remedy some flaws in C for writing C-style code (no, C++ was not supposed to be just C with a few tweaks).

The purpose of a (nonexperimental) programming language is to help build good systems. It follows that notions of system design and language design are closely related.

My definition of "good" in this context is basically "correct, maintainable, and providing acceptable resource usage." The obvious missing component is "easy to write," but for the kind of systems I think most about, that's secondary. "RAD development" is not my ideal. It can be as important to say what is not a primary aim as to state what is. For example, I have nothing against rapid development—nobody in their right mind wants to spend more time than necessary on a project—but I'd rather have lack of restrictions on application areas and performance. My aim for C++ was and is direct expression of ideas, resulting in code that can be efficient in time and space.

C and C++ have provided stability over decades. That has been immensely important to their industrial users. I have small programs that have been essentially unchanged since the early 80s. There is a price to pay for such stability, but languages that don't provide it are simply unsuitable for large, long-lived projects. Corporate languages and languages that try to follow trends closely tend to fail miserably here, causing a lot of misery along the way.

This leads to thinking about how to manage evolution. How much can be changed? What is the granularity of change? Changing a language every year or so as new releases of a product are released is too ad hoc and leads to a series of de facto subsets, discarded libraries and language features, and/or massive upgrade efforts. Also, a year is simply not sufficient gestation period for significant features, so the approach leads to half-baked solutions and dead ends. On the other hand, the 10-year cycle of ISO standardized languages, such as C and C++, is too long and leads to parts of the community (including parts of the committee) fossilizing.

A successful language develops a community: the community shares techniques, tools, and libraries. Corporate languages have an inherent advantage here: they can buy market share with marketing, conferences, and "free" libraries. This investment can pay off in terms of others adding significantly, making the community larger and more vibrant. Sun's efforts with Java showed how amateurish and underfinanced every previous effort to establish a (more or less) general-purpose language had been. The U.S. Department of Defense's efforts to establish Ada as a dominant language was a sharp contrast, as were the unfinanced efforts by me and my friends to establish C++.

I can't say that I approve of some of the Java tactics, such as selling top-down to nonprogramming executives, but it shows what can be done. Noncorporate successes include the Python and Perl communities. The successes at community building around C++ have been too few and too limited, given the size of the community. The ACCU conferences are great, but why haven't there been a continuous series of huge international C++ conferences since 1986 or so? The Boost libraries are great, but why hasn't there been a central repository for C++ libraries since 1986 or so? There are thousands of open source C++ libraries in use. I don't even know of a comprehensive list of commercial C++ libraries. I won't start answering those questions, but will just point out that any new language must somehow manage the centrifugal forces in a large community, or suffer pretty severe consequences.

A general-purpose language needs the input from and approval of several communities, such as, industrial programmers, educators, academic researchers, industrial researchers, and the open source community. These communities are not disjoint, but individual subcommunities often see themselves as self-sufficient, in possession of knowledge of what is right and in conflict with other communities that for some reason "don't get it." This can be a significant practical problem. For example, parts of the open source community have opposed the use of C++ because "it's a Microsoft language" (it isn't) or "AT&T owns it" (it doesn't), whereas some major industrial players have considered it a problem with C++ that *they* don't own it.

This really crucial problem here is that many subcommunities push a limited and parochial view of "what programming really is" and "what is really needed": "if everybody just did things the right way, there'd be no problem." The real problem is to balance the various needs to create a larger and more varied community. As people grow and face new challenges, the generality and flexibility of a language start to matter more than providing optimal solutions to a limited range of problems.

To get to technical points, I still think that a flexible, extensible, and general static type system is great. My reading of the C++ experience reinforces that view. I am also very keen on genuine local variables of user-defined types: the C++ techniques for handling general resources based on scoped variables have been very effective compared to just about anything. Constructors and destructors, often used together with RAII, can yield very elegant and efficient code.

Teaching

You left industry to become an academic. Why?

Bjarne: Actually, I haven't completely left industry, because I maintain a link to AT&T Labs as an AT&T fellow, and spend much time each year with industry people. I consider my connection with industry essential because that's what keeps my work anchored in reality.

I went to Texas A&M University as a professor five years ago because (after almost 25 years in "The Labs") I felt a need for a change and because I thought I had something to contribute in the area of education. I also entertained some rather idealistic ideas about doing more fundamental research after my years of very practical research and design.

Much computer science research is either too remote from everyday problems (even from conjectured future everyday problems), or so submerged in such everyday problems that it becomes little more than technology transfer. Obviously, I have nothing against technology transfer (we badly need it), but there ought to be strong feedback loops from industrial practice to advanced research. The short planning horizon of many in industry and the demands of the academic publication/tenure race conspire to divert attention and effort from some of the most critical problems.

During these years in academia, what did you learn about teaching programming to beginners?

Bjarne: The most concrete result of my years in academia (in addition to the obligatory academic papers) is a new textbook for teaching programming to people who have never programmed before, *Programming: Principles and Practice Using C++* [Addison-Wesley].

This is my first book for beginners. Before I went to academia, I simply didn't know enough beginners to write such a book. I did, however, feel that too many software developers were very poorly prepared for their tasks in industry and elsewhere. Now I have taught (and helped to teach) programming to more than 1,200 beginners and I feel a bit more certain that my ideas in this area can scale.

A beginner's book must serve several purposes. Most fundamentally, it must provide a good foundation for further learning (if successful, it will be the start of a lifelong effort) and also provide some practical skills. Also, programming—and in general software development—is not a purely theoretical skill, nor is it something you can do well without learning some fundamental concepts. Unfortunately, far too often, teaching fails to maintain a balance between theory/principles and practicalities/techniques. Consequently, we see people who basically despise programming ("mere coding") and think that software can be developed from first principles without any practical skills. Conversely, we see people who are convinced that "good code" is everything and can be achieved with little more than a quick look at an online manual and a lot of cutting and pasting; I have met programmers who considered K&R "too complicated and theoretical." My opinion is that both attitudes are far too extreme and lead to poorly structured, inefficient, and unmaintainable messes even when they do manage to produce minimally functioning code.

What is your opinion on code examples in textbooks? Should they include error/exception checking? Should they be complete programs so that they can actually be compiled and run?

Bjarne: I strongly prefer examples that in as few lines as possible illustrate an idea. Such program fragments are often incomplete, though I insist that mine will compile and run if embedded in suitable scaffolding code. Basically, my code presentation style is derived from K&R. For my new book, all code examples will be available in a compilable form. In the text, I vary between small fragments embedded in explanatory text and longer, more complete, sections of code. In key places, I use both techniques for a single example to allow the reader two looks at critical statements.

Some examples should be complete with error checking and all should reflect designs that can be checked. In addition to the discussion of errors and error handling scattered throughout the book, there are separate chapters on error handling and testing. I strongly prefer examples derived from real-world programs. I really dislike artificial cute examples, such as inheritance trees of animals and obtuse mathematical puzzles. Maybe I should add a label to my book: "no cute cuddly animals were abused in this book's examples."

CHAPTER TWO

Python

Python is a modern, general-purpose, high-level language developed by Guido van Rossum as a result of his work with the ABC programming language. Python's philosophy is pragmatic; its users often speak of the Zen of Python, strongly preferring a single obvious way to accomplish any task. Ports exist for VMs such as Microsoft's CLR and the JVM, but the primary implementation is CPython, still developed by van Rossum and other volunteers, who just released Python 3.0, a backward-incompatible rethinking of parts of the language and its core libraries.

The Pythonic Way

What differences are there between developing a programming language and developing a "common" software project?

Guido van Rossum: More than with most software projects, your most important users are programmers themselves. This gives a language project a high level of "meta" content. In the dependency tree of software projects, programming languages are pretty much at the bottom—everything else depends on one or more languages. This also makes it hard to change a language—an incompatible change affects so many dependents that it's usually just not feasible. In other words, all mistakes, once released, are cast in stone. The ultimate example of this is probably C++, which is burdened with compatibility requirements that effectively require code written maybe 20 years ago to be still valid.

How do you debug a language?

Guido: You don't. Language design is one area where agile development methodologies just don't make sense—until the language is stable, few people want to use it, and you won't find the bugs in the language definition until you have so many users that it's too late to change things.

Of course there's plenty in the *implementation* that can be debugged like any old program, but the language design itself pretty much requires careful design up front, because the cost of bugs is so exorbitant.

How do you decide when a feature should go in a library as an extension or when it needs to have support from the core language?

Guido: Historically, I've had a pretty good answer for that. One thing I noticed very early on was that everybody wants their favorite feature added to the language, and most people are relatively inexperienced about language design. Everybody is always proposing "let's add this to the language," "let's have a statement that does X." In many cases, the answer is, "Well, you can already do X or something almost like X by writing these two or three lines of code, and it's not all that difficult." You can use a dictionary, or you can combine a list and a tuple and a regular expression, or write a little metaclass—all of those things. I may even have had the original version of this answer from Linus, who seems to have a similar philosophy.

Telling people you can already do that and here is how is a first line of defense. The second thing is, "Well, that's a useful thing and we can probably write or you can probably write your own module or class, and encapsulate that particular bit of abstraction." Then the next line of defense is, "OK, this looks so interesting and useful that we'll actually accept it as a new addition to the standard library, and it's going to be pure Python." And then, finally, there are things that just aren't easy to do in pure Python and we'll suggest or recommend how to turn them into a C extension. The C extensions are the last line of defense before we have to admit, "Well, yeah, this is so useful and you really cannot do this, so we'll have to change the language."

There are other criteria that determine whether it makes more sense to add something to the language or it makes more sense to add something to the library, because if it has to do with the semantics of namespaces or that kind of stuff, there's really nothing you can do besides changing the language. On the other hand, the extension mechanism was made powerful enough that there is an amazing amount of stuff you can do from C code that extends the library and possibly even adds new built-in functionality without actually changing the language. The parser doesn't change. The parse tree doesn't change. The documentation for the language doesn't change. All your tools still work, and yet you have added new functionality to your system.

I suppose there are probably features that you've looked at that you couldn't implement in Python other than by changing the language, but you probably rejected them. What criteria do you use to say this is something that's Pythonic, this is something that's not Pythonic?

Guido: That's much harder. That is probably, in many cases, more a matter of a gut feeling than anything else. People use the word Pythonic and "that is Pythonic" a lot, but nobody can give you a watertight definition of what it means for something to be Pythonic or un-Pythonic.

You have the "Zen of Python," but beyond that?

Guido: That requires a lot of interpretation, like every good holy book. When I see a good or a bad proposal, I can tell if it is a good or bad proposal, but it's really hard to write a set of rules that will help someone else to distinguish good language change proposals from bad change proposals.

Sounds almost like it's a matter of taste as much as anything.

Guido: Well, the first thing is always try to say "no," and see if they go away or find a way to get their itch scratched without changing the language. It's remarkable how often that works. That's more of a operational definition of "it's not necessary to change the language."

If you keep the language constant, people will still find a way to do what they need to do. Beyond that it's often a matter of use cases coming from different areas where there is nothing application-specific. If something was really cool for the Web, that would not make it a good feature to add to the language. If something was really good for writing shorter functions or writing classes that are more maintainable, that might be a good thing to add to the language. It really needs to transcend application domains in general, and make things simpler or more elegant.

When you change the language, you affect everyone. There's no feature that you can hide so well that most people don't need to know about. Sooner or later, people will encounter code written by someone else that uses it, or they'll encounter some obscure corner case where they have to learn about it because things don't work the way they expected.

Often elegance is also in the eye of the beholder. We had a recent discussion on one of the Python lists where people were arguing forcefully that using `dollar` instead of `self-dot` was much more elegant. I think their definition of elegance was number of keystrokes.

There's an argument to make for parsimony there, but very much in the context of personal taste.

Guido: Elegance and simplicity and generality all are things that, to a large extent, depend on personal taste, because what seems to cover a larger area for me may not cover enough for someone else, and vice versa.

How did the Python Enhancement Proposal (PEP) process come about?

Guido: That's a very interesting historical tidbit. I think it was mostly started and championed by Barry Warsaw, one of the core developers. He and I started working together in '95, and I think around 2000, he came up with the suggestion that we needed more of a formal process around language changes.

I tend to be slow in these things. I mean I wasn't the person who discovered that we really needed a mailing list. I wasn't the person who discovered that the mailing list got unwieldy and we needed a newsgroup. I wasn't the person to propose that we needed a website. I was also not the person to propose that we needed a process for discussing and inventing language changes, and making sure to avoid the occasional mistake where things had been proposed and quickly accepted without thinking through all of the consequences.

At the time between 1995 and 2000, Barry, myself, and a few other core developers, Fred Drake, Ken Manheimer for a while, were all at CNRI, and one of the things that CNRI did was organize the IETF meetings. CNRI had this little branch that eventually split off that was a conference organizing bureau, and their only customer was the IETF. They later also did the Python conferences for a while, actually. Because of that it was a pretty easy boondoggle to attend IETF meetings even if they weren't local. I certainly got a taste of the IETF process with its RFCs and its meeting groups and stages, and Barry also got a taste of that. When he proposed to do something similar for Python, that was an easy argument to make. We consciously decided that we wouldn't make it quite as heavy-handed as the IETF RFCs had become by then, because Internet standards, at least some of them, affect way more industries and people and software than a Python change, but we definitely modeled it after that. Barry is a genius at coming up with good names, so I am pretty sure that PEP was his idea.

We were one of the first open source projects at the time to have something like this, and it's been relatively widely copied. The Tcl/Tk community basically changed the title and used exactly the same defining document and process, and other projects have done similar things.

Do you find that adding a little bit of formalism really helps crystallize the design decisions around Python enhancements?

Guido: I think it became necessary as the community grew and I wasn't necessarily able to judge every proposal on its value by itself. It has really been helpful for me to let other people argue over various details, and then come with relatively clear-cut conclusions.

Do they lead to a consensus where someone can ask you to weigh in on a single particular crystallized set of expectations and proposals?

Guido: Yes. It often works in a way where I initially give a PEP a thumb's up in the sense that I say, "It looks like we have a problem here. Let's see if someone figures out what the right solution is." Often they come out with a bunch of clear conclusions on how the problem should be solved and also a bunch of open issues. Sometimes my gut feelings can help close the open issues. I'm very active in the PEP process when it's an area that I'm excited about—if we had to add a new loop control statement, I wouldn't want that to be designed by other people. Sometimes I stay relatively far away from it like database APIs.

What creates the need for a new major version?

Guido: It depends on your definition of major. In Python, we generally consider releases like 2.4, 2.5, and 2.6 "major" events, which only happen every 18–24 months. These are the only occasions where we can introduce new features. Long ago, releases were done at the whim of the developers (me, in particular). Early this decade, however, the users requested some predictability—they objected against features being added or changed in "minor" revisions (e.g., 1.5.2 added major features compared to 1.5.1), and they wished the major releases to be supported for a certain minimum amount of time (18 months). So now we have more or less time-based major releases: we plan the series of dates leading up to a major release (e.g., when alpha and beta versions and release candidates are issued) long in advance, based on things like release manager availability, and we urge the developers to get their changes in well in advance of the final release date.

Features selected for addition to releases are generally agreed upon by the core developers, after (sometimes long) discussions on the merits of the feature and its precise specification. This is the PEP process: Python Enhancement Proposal, a document-base process not unlike the IETF's RFC process or the Java world's JSR process, except that we aren't quite as formal, as we have a much smaller community of developers. In case of prolonged disagreement (either on the merits of a feature or on specific details), I may end up breaking a tie; my tie-breaking algorithm is mostly intuitive, since by the time it is invoked, rational argument has long gone out of the window.

The most contentious discussions are typically about user-visible language features; library additions are usually easy (as they don't harm users who don't care), and internal improvements are not really considered features, although they are constrained by pretty stringent backward compatibility at the C API level.

Since the developers are typically the most vocal users, I can't really tell whether features are proposed by users or by developers—in general, developers propose features based on needs they perceived among the users they know. If a user proposes a new feature, it is rarely a success, since without a thorough understanding of the implementation (and of language design and implementation in general) it is nearly impossible to properly propose a new feature. We like to ask users to explain their problems without having a specific solution in mind, and then the developers will propose solutions and discuss the merits of different alternatives with the users.

There's also the concept of a radically major or breakthrough version, like 3.0. Historically, 1.0 was evolutionarily close to 0.9, and 2.0 was also a relatively small step from 1.6. From now on, with the much larger user base, such versions are rare indeed, and provide the only occasion for being truly incompatible with previous versions. Major versions are made backward compatible with previous major versions with a specific mechanism available for deprecating features slated for removal.

How did you choose to handle numbers as arbitrary precision integers (with all the cool advantages you get) instead of the old (and super common) approach to pass it to the hardware?

Guido: I originally inherited this idea from Python's predecessor, ABC. ABC used arbitrary precision rationals, but I didn't like the rationals that much, so I switched to integers; for reals, Python uses the standard floating-point representation supported by the hardware (and so did ABC, with some prodding).

Originally Python had two types of integers: the customary 32-bit variety ("int") and a separate arbitrary precision variety ("long"). Many languages do this, but the arbitrary precision variety is relegated to a library, like Bignum in Java and Perl, or GNU MP for C. In Python, the two have (nearly) always lived side-by-side in the core language, and users had to choose which one to use by appending an "L" to a number to select the long variety. Gradually this was considered an annoyance; in Python 2.2, we introduced automatic conversion to long when the mathematically correct result of an operation on ints could not be represented as an int (for example, 2**100).

Previously, this would raise an `OverflowError` exception. There was once a time where the result would silently be truncated, but I changed it to raising an exception before ever letting others use the language. In early 1990, I wasted an afternoon debugging a short demo program I'd written implementing an algorithm that made non-obvious use of very large integers. Such debugging sessions are seminal experiences.

However, there were still certain cases where the two number types behaved slightly different; for example, printing an int in hexadecimal or octal format would produce an unsigned outcome (e.g., –1 would be printed as FFFFFFFF), while doing the same on the mathematically equal long would produce a signed outcome (–1, in this case). In Python 3.0, we're taking the radical step of supporting only a single integer type; we're calling it `int`, but the implementation is largely that of the old long type.

Why do you call it a radical step?

Guido: Mostly because it's a big deviation from current practice in Python. There was a lot of discussion about this, and people proposed various alternatives where two (or more) representations would be used internally, but completely or mostly hidden from end users (but not from C extension writers). That might perform a bit better, but in the end it was already a massive amount of work, and having two representations internally would just increase the effort of getting it right, and make interfacing to it from C code even hairier. We are now hoping that the performance hit is minor and that we can improve performance with other techniques like caching.

How did you adopt the "there should be one—and preferably only one—obvious way to do it" philosophy?

Guido: This was probably subconscious at first. When Tim Peters wrote the "Zen of Python" (from which you quote), he made explicit a lot of rules that I had been applying without being aware of them. That said, this particular rule (while often violated, with my consent) comes straight from the general desire for elegance in mathematics and computer science. ABC's authors also applied it, in their desire for a small number of orthogonal types or concepts. The idea of orthogonality is lifted straight from mathematics, where it refers to the very *definition* of having one way (or one true way) to express something. For example, the XYZ coordinates of any point in 3D space are uniquely determined, once you've picked an origin and three basis vectors.

I also like to think that I'm doing most users a favor by not requiring them to choose between similar alternatives. You can contrast this with Java, where if you need a listlike data structure, the standard library offers many versions (a linked list, or an array list, and others), or C, where you have to decide how to implement your own list data type.

What is your take on static versus dynamic typing?

Guido: I wish I could say something simple like "static typing bad, dynamic typing good," but it isn't always that simple. There are different approaches to dynamic typing, from Lisp to Python, and different approaches to static typing, from C++ to Haskell. Languages like C++ and Java probably give static typing a bad name because they require you to tell the compiler the same thing several times over. Languages like Haskell and ML, however, use type inferencing, which is quite different, and has some of the same benefits as dynamic typing, such as more concise expression of ideas in code. However the functional paradigm seems to be hard to use on its own—things like I/O or GUI interaction don't fit well into that mold, and typically are solved with the help of a bridge to a more traditional language, like C, for example.

In some situations the verbosity of Java is considered a plus; it has enabled the creation of powerful code-browsing tools that can answer questions like "where is this variable changed?" or "who calls this method?" Dynamic languages make answering such questions harder, because it's often hard to find out the type of a method argument without analyzing every path through the entire codebase. I'm not sure how functional languages

like Haskell support such tools; it could well be that you'd have to use essentially the same technique as for dynamic languages, since that's what type inferencing does anyway—in my limited understanding!

Are we moving toward hybrid typing?

Guido: I expect there's a lot to say for some kind of hybrid. I've noticed that most large systems written in a statically typed language actually contain a significant subset that is essentially dynamically typed. For example, GUI widget sets and database APIs for Java often feel like they are fighting the static typing every step of the way, moving most correctness checks to runtime.

A hybrid language with functional and dynamic aspects might be quite interesting. I should add that despite Python's support for some functional tools like `map()` and `lambda`, Python does *not* have a functional-language subset: there is no type inferencing, and no opportunity for parallellization.

Why did you choose to support multiple paradigms?

Guido: I didn't really; Python supports procedural programming, to some extent, and OO. These two aren't so different, and Python's procedural style is still strongly influenced by objects (since the fundamental data types are all objects). Python supports a tiny bit of functional programming—but it doesn't resemble any real functional language, and it never will. Functional languages are all about doing as much as possible at compile time—the "functional" aspect means that the compiler can optimize things under a very strong guarantee that there are no side effects, unless explicitly declared. Python is about having the simplest, dumbest compiler imaginable, and the official runtime semantics actively discourage cleverness in the compiler like parallelizing loops or turning recursion into loops.

Python probably has the reputation of supporting functional programming based on the inclusion of `lambda`, `map`, `filter`, and `reduce` in the language, but in my eyes these are just syntactic sugar, and not the fundamental building blocks that they are in functional languages. The more fundamental property that Python shares with Lisp (not a functional language either!) is that functions are first-class objects, and can be passed around like any other object. This, combined with nested scopes and a generally Lisp-like approach to function state, makes it possible to easily implement concepts that superficially resemble concepts from functional languages, like currying, map, and reduce. The primitive operations that are necessary to *implement* those concepts are built in Python, where in functional languages, those concepts *are* the primitive operations. You can write `reduce()` in a few lines of Python. Not so in a functional language.

When you created the language, did you consider the type of programmers it might have attracted?

Guido: Yes, but I probably didn't have enough imagination. I was thinking of professional programmers in a Unix or Unix-like environment. Early versions of the Python tutorial used a slogan something like "Python bridges the gap between C and shell programming,"

because that was where I was myself, and the people immediately around me. It never occurred to me that Python would be a good language to embed in applications until people started asking about that.

The fact that it was useful for teaching first principles of programming in a middle school or college setting or for self-teaching was merely a lucky coincidence, enabled by the many ABC features that I kept—ABC was aimed specifically at teaching programming to nonprogrammers.

How do you balance the different needs of a language that should be easy to learn for novices versus a language that should be powerful enough for experienced programmers to do useful things? Is that a false dichotomy?

Guido: Balance is the word. There are some well-known traps to avoid, like stuff that is thought to help novices but annoys experts, and stuff that experts need but confuses novices. There's plenty enough space in between to keep both sides happy. Another strategy is to have ways for experts to do advanced things that novices will never encounter—for example, the language supports metaclasses, but there's no reason for novices to know about them.

The Good Programmer

How do you recognize a good programmer?

Guido: It takes time to recognize a good programmer. For example, it's really hard to tell good from bad in a one-hour interview. When you work together with someone though, on a variety of problems, it usually becomes pretty clear which are the good ones. I hesitate to give specific criteria—I guess in general the good ones show creativity, learn quickly, and soon start producing code that works and doesn't need a lot of changes before it's ready to be checked in. Note that some folks are good at different aspects of programming than others—some folks are good at algorithms and data structures, others are good at large-scale integration, or protocol design, or testing, or API design, or user interfaces, or whatever other aspects of programming exist.

What method would you use to hire programmers?

Guido: Based on my interviewing experience in the past, I don't think I'd be any good at hiring in the traditional way—my interview skills are nearly nonexistent on both sides of the table! I guess what I'd do would be to use some kind of apprentice system where I'd be working closely with people for quite some time and would eventually get a feeling for their strengths and weaknesses. Sort of the way an open source project works.

Is there any characteristic that becomes fundamental to evaluate if we are looking for great Python programmers?

Guido: I'm afraid you are asking this from the perspective of the typical manager who simply wants to hire a bunch of Python programmers. I really don't think there's a simple answer, and in fact I think it's probably the wrong question. You don't want to hire Python programmers. You want to hire smart, creative, self-motivated people.

If you check job ads for programmers, nearly all of them include a line about being able to work in a team. What is your opinion on the role of the team in programming? Do you still see space for the brilliant programmer who can't work with others?

Guido: I am with the job ads in that one aspect. Brilliant programmers who can't do teamwork shouldn't get themselves in the position of being hired into a traditional programming position—it will be a disaster for all involved, and their code will be a nightmare for whoever inherits it. I actually think it's a distinct lack of brilliance if you can't do teamwork. Nowadays there are ways to learn how to work with other people, and if you're really so brilliant you should be able to learn teamwork skills easily—it's really not as hard as learning how to implement an efficient Fast Fourier Transform, if you set your mind about it.

Being the designer of Python, what advantages do you see when coding with your language compared to another skilled developer using Python?

Guido: I don't know—at this point the language and VM have been touched by so many people that I'm sometimes surprised at how certain things work in detail myself! If I have an advantage over other developers, it probably has more to do with having used the language longer than anyone than with having written it myself. Over that long period of time, I have had the opportunity to ponder which operations are faster and which are slower—for example, I may be aware more than most users that locals are faster than globals (though *others* have gone overboard using this, not me!), or that functions and method calls are expensive (more so than in C or Java), or that the fastest data type is a tuple.

When it comes to using the standard library and beyond, I often feel that others have an advantage. For example, I write about one web application every few years, and the technology available changes each time, so I end up writing a "first" web app using a new framework or approach each time. And I still haven't had the opportunity to do serious XML mangling in Python.

It seems that one of the features of Python is its conciseness. How does this affect the maintainability of the code?

Guido: I've heard of research as well as anecdotal evidence indicating that the error rate per number of lines of code is pretty consistent, regardless of the programming language used. So a language like Python where a typical application is just much smaller than, say, the same amount of functionality written in C++ or Java, would make that application much more maintainable. Of course, this is likely going to mean that a single programmer is responsible for more functionality. That's a separate issue, but it still comes out in favor of Python: more productivity per programmer probably means fewer programmers on a team, which means less communication overhead, which according to *The Mythical Man-Month* [Frederick P. Brooks; Addison-Wesley Professional] goes up by the square of the team size, if I remember correctly.

What link do you see between the easiness of prototyping offered by Python and the effort needed to build a complete application?

Guido: I never meant Python to be a prototyping language. I don't believe there should be a clear distinction between prototyping and "production" languages. There are situations where the best way to write a prototype would be to write a little throwaway C hack. There are other situations where a prototype can be created using no "programming" at all—for example, using a spreadsheet or a set of `find` and `grep` commands.

The earliest intentions I had for Python were simply for it to be a language to be used in cases where C was overkill and shell scripts became too cumbersome. That covers a lot of prototyping, but it also covers a lot of "business logic" (as it's come to be called these days) that isn't particularly greedy in computing resources but requires a lot of code to be written. I would say that most Python code is not written as a prototype but simply to get a job done. In most cases Python is fully up to the job, and there is no need to change much in order to arrive at the final application.

A common process is that a simple application gradually acquires more functionality, and ends up growing tenfold in complexity, and there is never a precise cutover point from prototype to final application. For example, the code review application Mondrian that I started at Google has probably grown tenfold in code size since I first released it, and it is still all written in Python. Of course, there are also examples where Python did eventually get replaced by a faster language—for example, the earliest Google crawler/indexer was (largely) written in Python—but those are the exceptions, not the rule.

How does the immediacy of Python affect the design process?

Guido: This is often how I work, and, at least for me, in general it works out well! Sure, I write a lot of code that I throw away, but it's much less code than I would have written in any other language, and writing code (without even running it) often helps me tremendously in understanding the details of the problem. Thinking about how to rearrange the code so that it solves the problem in an optimal fashion often helps me think about the problem. Of course, this is not to be used as an excuse to avoid using a whiteboard to sketch out a design or architecture or interaction, or other early design techniques. The trick is to use the right tool for the job. Sometimes that's a pencil and a napkin—other times it's an Emacs window and a shell prompt.

Do you think that bottom-up program development is more suited to Python?

Guido: I don't see bottom-up versus top-down as religious opposites like vi versus Emacs. In any software development process, there are times when you work bottom-up, and other times when you work top-down. Top-down probably means you're dealing with something that needs to be carefully reviewed and designed before you can start coding, while bottom-up probably means that you are building new abstractions on top of existing ones, for example, creating new APIs. I'm not implying that you should start coding APIs without having some kind of design in mind, but often new APIs follow logically from the available lower-level APIs, and the design work happens while you are actually writing code.

When do you think Python programmers appreciate more its dynamic nature?

Guido: The language's dynamic features are often most useful when you are exploring a large problem or solution space and you don't know your way around yet—you can do a bunch of experiments, each using what you learned from the previous ones, without having too much code that locks you into a particular approach. Here it really helps that you can write very compact code in Python—writing 100 lines of Python to run an experiment once and then starting over is much more efficient than writing a 1,000-line framework for experimentation in Java and then finding out it solves the wrong problem!

From a security point of view, what does Python offer to the programmer?

Guido: That depends on the attacks you're worried about. Python has automatic memory allocation, so Python programs aren't prone to certain types of bugs that are common in C and C++ code like buffer overflows or using deallocated memory, which have been the bread and butter of many attacks on Microsoft software. Of course the Python runtime itself is written in C, and indeed vulnerabilities have been found here over the years, and there are intentional escapes from the confines of the Python runtime, like the `ctypes` module that lets one call arbitrary C code.

Does its dynamic nature help or rather the opposite?

Guido: I don't think the dynamic nature helps or hurts. One could easily design a dynamic language that has lots of vulnerabilities, or a static language that has none. However having a runtime, or *virtual machine* as is now the "hip" term, helps by constraining access to the raw underlying machine. This is coincidentally one of the reasons that Python is the first language supported by Google App Engine, the project in which I am currently participating.

How can a Python programmer check and improve his code security?

Guido: I think Python programmers shouldn't worry much about security, certainly not without having a specific attack model in mind. The most important thing to look for is the same as in all languages: be suspicious of data provided by someone you don't trust (for a web server, this is every byte of the incoming web request, even the headers). One specific thing to watch out for is regular expressions—it is easy to write a regular expression that runs in exponential time, so web applications that implement searches where the end user types in a regular expression should have some mechanism to limit the running time.

Is there any fundamental concept (general rule, point of view, mindset, principle) that you would suggest to be proficient in developing with Python?

Guido: I would say pragmatism. If you get too hung up about theoretical concepts like data hiding, access control, abstractions, or specifications, you aren't a real Python programmer, and you end up wasting time fighting the language, instead of using (and enjoying) it; you're also likely to use it inefficiently. Python is good if you're an instant gratification junkie like myself. It works well if you enjoy approaches like extreme programming or

other agile development methods, although even there I would recommend taking everything in moderation.

What do you mean by "fighting the language"?

Guido: That usually means that they're trying to continue their habits that worked well with a different language.

A lot of the proposals to somehow get rid of explicit self come from people who have recently switched to Python and still haven't gotten used to it. It becomes an obsession for them. Sometimes they come out with a proposal to change the language; other times they come up with some super-complicated metaclass that somehow makes self implicit. Usually things like that are super-inefficient or don't actually work in a multithreaded environment or whatever other edge case, or they're so obsessed about having to type those four characters that they changed the convention from self to `s` or capital `S`. People will turn everything into a class, and turn every access into an accessor method, where that is really not a wise thing to do in Python; you'll just have more verbose code that is harder to debug and runs a lot slower. You know the expression "You can write FORTRAN in any language?" You can write Java in any language, too.

You spent so much time trying to create (preferably) one obvious way to do things. It seems like you're of the opinion that doing things that way, the Python way, really lets you take advantage of Python.

Guido: I'm not sure that I really spend a lot of time making sure that there's only one way. The "Zen of Python" is much younger than the language Python, and most defining characteristics of the language were there long before Tim Peters wrote it down as a form of poetry. I don't think he expected it to be quite as widespread and successful when he wrote it up.

It's a catchy phrase.

Guido: Tim has a way with words. "There's only one way to do it" is actually in most cases a white lie. There are many ways to do data structures. You can use tuples and lists. In many cases, it really doesn't matter that much whether you use a tuple or a list or sometimes a dictionary. It turns out usually if you look really carefully, one solution is objectively better because it works just as well in a number of situations, and there's one or two cases where lists just works so much better than tuples when you keep growing them.

That comes more actually from the original ABC philosophy that was trying to be very sparse in the components. ABC actually shared a philosophy with ALGOL-68, which is now one of the deadest languages around, but was very influential. Certainly where I was at the time during the 80s, it was very influential because Adriaan van Wijngaarden was the big guy from ALGOL 68. He was still teaching classes when I went to college. I did one or two semesters where he was just telling anecdotes from the history of ALGOL 68 if he felt like it. He had been the director of CWI. Someone else was it by the time I joined.

There were many people who had been very close with ALGOL 68. I think Lambert Meertens, the primary author of ABC, was also one of the primary editors of the ALGOL 68 report, which probably means he did a lot of the typesetting, but he may occasionally also have done quite a lot of the thinking and checking. He was clearly influenced by ALGOL 68's philosophy of providing constructs that can be combined in many different ways to produce all sorts of different data structures or ways of structuring a program.

It was definitely his influence that said, "We have lists or arrays, and they can contain any kind of other thing. They can contain numbers or strings, but they can also contain other arrays and tuples of other things. You can combine all of these things together." Suddenly you don't need a separate concept of a multidimensional array because an array of arrays solves that for any dimensionality. That philosophy of taking a few key things that cover different directions of flexibility and allow them to be combined was very much a part of ABC. I borrowed all of that almost without thinking about it very hard.

While Python tries to give the appearance that you can combine things in very flexible ways as long as you don't try to nest statements inside expressions, there is actually a remarkable number of special cases in the syntax where in some cases a comma means a separation between parameters, and in other cases the comma means the items of a list, and in yet another case it means an implicit tuple.

There are a whole bunch of variations in the syntax where certain operators are not allowed because they would conflict with some surrounding syntax. That is never really a problem because you can always put an extra pair of parentheses around something when it doesn't work. Because of that the syntax, at least from the parser author's perspective, has grown quite a bit. Things like list comprehensions and generator expressions are syntactically still not completely unified. In Python 3000, I believe they are. There's still some subtle semantic differences, but the syntax at least is the same.

Multiple Pythons

Does the parser get simpler in Python 3000?

Guido: Hardly. It didn't become more complex, but it also didn't really become simpler.

No more complex I think is a win.

Guido: Yeah.

Why the simplest, dumbest compiler imaginable?

Guido: That was originally a very practical goal, because I didn't have a degree in code generation. There was just me, and I had to have the byte code generator behind me before I could do any other interesting work on the language.

I still believe that having a very simple parser is a good thing; after all, it is just the thing that turns the text into a tree that represents the structure of the program. If the syntax is so ambiguous that it takes really advanced parts of technology to figure it out, then human readers are probably confused half the time as well. It also makes it really hard to write another parser.

Python is incredibly simple to parse, at least at the syntactic level. At the lexical level, the analysis is relatively subtle because you have to read the indentation with a little stack that is embedded in the lexical analyzer, which is a counterexample for the theory of separation between lexical and grammatical analysis. Nevertheless, that is the right solution. The funny thing is that I love automatically generated parsers, but I do not believe very strongly in automatically generated lexical analysis. Python has always had a manually generated scanner and an automated parser.

People have written many different parsers for Python. Every port of Python to a different virtual machine, whether Jython or IronPython or PyPy, has its own parser, and it's no big deal because the parser is never a very complex piece of the project, because the structure of the language is such that you can very easily parse it with the most basic one-token lookahead recursive descent parser.

What makes parsers slow is actually ambiguities that can only be resolved by looking ahead until the end of the program. In natural languages there are many examples where it's impossible to parse a sentence until you've read the last word and the arbitrary nesting in the sentence. Or there are sentences that can only be parsed if you actually know the person that they are talking about, but that's a completely different situation. For parsing programming languages, I like my one-token lookahead.

That suggests to me that there may never be macros in Python because you have to perform another parsing phase then!

Guido: There are ways of embedding the macros in the parser that could probably work. I'm not at all convinced that macros solve any problem that is particularly pressing for Python, though. On the other hand, since the language is easy to parse, if you come up with some kind of hygienic set of macros that fit within the language syntax, it might be very simple to implement micro-evaluation as parse tree manipulations. That's just not an area that I'm particularly interested in.

Why did you choose to use strict formatting in source code?

Guido: The choice of indentation for grouping was not a novel concept in Python; I inherited this from ABC, but it also occurred in occam, an older language. I don't know if the ABC authors got the idea from occam, or invented it independently, or if there was a common ancestor. The idea may be attributed to Don Knuth, who proposed this as early as 1974.

Of course, I could have chosen not to follow ABC's lead, as I did in other areas (e.g., ABC used uppercase for language keywords and procedure names, an idea I did not copy), but I had come to like the feature quite a bit while using ABC, as it seemed to do away with a certain type of pointless debate common amongst C users at the time, about where to place the curly braces. I also was well aware that readable code uses indentation voluntarily anyway to indicate grouping, and I had come across subtle bugs in code where the indentation disagreed with the syntactic grouping using curly braces—the programmer and any reviewers had assumed that the indentation matched the grouping and therefore not noticed the bug. Again, a long debugging session taught a valuable lesson.

Strict formatting should produce a cleaner code and probably reduce the differences in the "layout" of the code of different programmers, but doesn't this sound like forcing a human being to adapt to the machine, instead of the opposite path?

Guido: Quite the contrary—it helps the human reader more than it helps the machine; see the previous example. Probably the advantages of this approach are more visible when maintaining code written by another programmer.

New users are often put off by this initially, although I don't hear about this so much any more; perhaps the people teaching Python have learned to anticipate this effect and counter it effectively.

I would like to ask you about multiple implementations of Python. There are four or five big implementations, including Stackless and PyPy.

Guido: Stackless, technically, is not a separate implementation. Stackless is often listed as a separate Python implementation because it is a fork of Python that replaces a pretty small part of the virtual machine with a different approach.

Basically the byte code dispatch, right?

Guido: Most of the byte code dispatch is very similar. I think the byte codes are the same and certainly all of the objects are the same. What they do different is when you have a call from one Python procedure to another procedure: they do that with manipulation of objects, where they just push a stack of stack frames and the same bit of C code remains in charge. The way it's done in C Python is that, at that point, a C function is invoked which will then eventually invoke a new instance of the virtual machine. It's not really the whole virtual machine, but the loop that interprets the byte code. There's only one of those loops on the C stack in stackless. In traditional C Python, you can have that same loop on your C stack many times. That's the only difference.

PyPy, IronPython, Jython are separate implementations. I don't know about something that translates to JavaScript, but I wouldn't be surprised if someone had gotten quite far with that at some point. I have heard of experimental things that translate to OCaml and Lisp and who knows what. There once was something that translated to C code as well.

Mark Hammond and Greg Stein worked on it in the late 90s, but they found out that the speedup that they could obtain was very, very modest. In the best circumstances, it would run twice as fast; also, the generated code was so large that you had these enormous binaries, and that became a problem.

Start-up time hurt you there.

Guido: I think the PyPy people are on the right track.

It sounds like you're generally supportive of these implementations.

Guido: I have always been supportive of alternate implementations. From the day that Jim Hugunin walked in the door with a more or less completed JPython implementation, I was excited about it. In a sense, it counts as a validation of the language design. It also means that people can use their favorite language on the platform where otherwise they wouldn't have access to it. We still have a way to go there, but it certainly helped me isolate which features were really features of the language that I cared about, and which features were features of a particular implementation where I was OK with other implementations doing things differently. That's where we ended up on the unfortunately slippery slope of garbage collection.

That's always a slippery slope.

Guido: But it's also necessary. I cannot believe how long we managed to live with pure reference counting and no way to break cycles. I have always seen reference counting as a way of doing garbage collection, and not a particularly bad one. There used to be this holy war between reference counting versus garbage collection, and that always seemed rather silly to me.

Regarding these implementations again, I think Python is an interesting space because it has a pretty good specification. Certainly compared to other languages like Tcl, Ruby, and Perl 5. Was that something that came about because you wanted to standardize the language and its behavior, or because you were looking at multiple implementations, or something else?

Guido: It was probably more a side effect of the community process around PEPs and the multiple implementations. When I originally wrote the first set of documentation, I very enthusiastically started a language reference manual, which was supposed to be a sufficiently precise specification that someone from Mars or Jupiter could implement the language and get the semantics right. I never got anywhere near fulfilling that goal.

ALGOL 68 probably got the closest of any language ever with their highly mathematical specification. Other languages like C++ and JavaScript have managed with sheer willpower of the standardization committee, especially in the case of C++. That's obviously an incredibly impressive effort. At the same time, it takes so much manpower to write a specification that is that precise, that my hope of getting something like that for Python never really got implemented.

What we do have is enough understanding of how the language is supposed to work, and enough unit tests, and enough people on hand that can answer to implementers of other versions in finite time. I know that, for example, the IronPython folks have been very conscientious in trying to run the entire Python test suite, and for every failure deciding if the test suite was really testing the specific behavior of the C Python implementation or if they actually had more work to do in their implementation.

The PyPy folks did the same thing, and they went one step further. They have a couple of people who are much smarter than I, and who have come up with an edge case probably prompted by their own thinking about how to generate code and how to analyze code in a JIT environment. They have actually contributed quite a few tests and disambiguations and questions when they found out that there was a particular combination of things that nobody had ever really thought about. That was very helpful. The process of having multiple implementations of the language has been tremendously helpful for getting the specification of the language disambiguated.

Do you foresee a time when C Python may not be the primary implementation?

Guido: That's hard to see. I mean some people foresee a time where .NET rules the world; other people foresee a time where JVMs rule the world. To me, that all seems like wishful thinking. At the same time, I don't know what will happen. There could be a quantum jump where, even though the computers that we know don't actually change, a different kind of platform suddenly becomes much more prevalent and the rules are different.

Perhaps a shift away from the von Neumann architecture?

Guido: I wasn't even thinking of that, but that's certainly also a possibility. I was more thinking of what if mobile phones become the ubiquitous computing device. Mobile phones are only a few years behind the curve of the power of regular laptops, which suggests that in a few years, mobile phones, apart from the puny keyboard and screen, will have enough computing power so that you don't need a laptop anymore. It may well be that mobile phones for whatever platform politics end up all having a JVM or some other standard environment where C Python is not the best approach and some other Python implementation would work much better.

There's certainly also the question of what do we do when we have 64 cores on a chip, even in a laptop or in a cell phone. I don't actually know if that should change the programming paradigm all that much for most of the things we do. There may be a use for some languages that let you specify incredibly subtle concurrent processes, but in most cases the average programmer cannot write correct thread-safe code anyway. Assuming that somehow the ascent of multiple cores forces them to do that is kind of unrealistic. I expect that multiple cores will certainly be useful, but they will be used for coarse-grained parallelism, which is better anyway, because with the enormous cost difference between cache hits and cache misses, main memory no longer really serves the function of shared memory. You want to have your processes as isolated as possible.

How should we deal with concurrency? At what level should this problem be dealt with or, even better, solved?

Guido: My feeling is that writing single-threaded code is hard enough, and writing multi-threaded code is way harder—so hard that most people don't have a hope of getting it right, and that includes myself. Therefore, I don't believe that fine-grained synchronization primitives and shared memory are the solution—instead, I'd much rather see message-passing solutions get back in style. I'm pretty sure that changing all programming languages to add synchronization constructs is a bad idea.

I also still don't believe that trying to remove the GIL from CPython will work. I do believe that some support for managing multiple processes (as opposed to threads) is a piece of the puzzle, and for that reason Python 2.6 and 3.0 will have a new standard library module, multiprocessing, that offers an API similar to that of the threading module for doing exactly that. As a bonus, it even supports processes running on different hosts!

Expedients and Experience

Is there any tool or feature that you feel is missing when writing software?

Guido: If I could sketch on a computer as easily as I can with pencil and paper, I might be making more sketches while doing the hard thinking about a design. I fear that I'll have to wait until the mouse is universally replaced by a pen (or your finger) that lets you draw on the screen. Personally, I feel terribly handicapped when using any kind of computerized drawing tool, even if I'm pretty good with pencil and paper—perhaps I inherited it from my father, who was an architect and was always making rough sketches, so I was always sketching as a teenager.

At the other end of the scale, I suppose I may not even know what I'm missing for spelunking large codebases. Java programmers have IDEs now that provide quick answers to questions like "where are the callers of this method?" or "where is this variable assigned to?" For large Python programs, this would also be useful, but the necessary static analysis is harder because of Python's dynamic nature.

How do you test and debug your code?

Guido: Whatever is expedient. I do a lot of testing when I write code, but the testing method varies per project. When writing your basic pure algorithmic code, unit tests are usually great, but when writing code that is highly interactive or interfaces to legacy APIs, I often end up doing a lot of manual testing, assisted by command-line history in the shell or page-reload in the browser. As an (extreme) example, you can't very well write a unit test for a script whose sole purpose is to shut down the current machine; sure, you can mock out the part that actually does the shut down, but you still have to test that part, too, or else how do you know that your script actually works?

Testing something in different environments is also often hard to automate. Buildbot is great for large systems, but the overhead to set it up is significant, so for smaller systems often you just end up doing a lot of manual QA. I've gotten a pretty good intuition for doing QA, but unfortunately it's hard to explain.

When should debugging be taught? And how?

Guido: Continuously. You are debugging your entire life. I just "debugged" a problem with my six-year-old son's wooden train set where his trains kept getting derailed at a certain point on the track. Debugging is usually a matter of moving down an abstraction level or two, and helped by stopping to look carefully, thinking, and (sometimes) using the right tools.

I don't think there is a single "right" way of debugging that can be taught at a specific point, even for a very specific target such as debugging program bugs. There is an incredibly large spectrum of possible causes for program bugs, including simple typos, "thinkos," hidden limitations of underlying abstractions, and outright bugs in abstractions or their implementation. The right approach varies from case to case. Tools come into play mostly when the required analysis ("looking carefully") is tedious and repetitive. I note that Python programmers often need few tools because the search space (the program being debugged) is so much smaller.

How do you resume programming?

Guido: This is actually an interesting question. I don't recall ever looking consciously at how I do this, while I indeed deal with this all the time. Probably the tool I used most for this is version control: when I come back to a project I do a diff between my workspace and the repository, and that will tell me the state I'm in.

If I have a chance, I leave XXX markers in the unfinished code when I know I am about to be interrupted, telling me about specific subtasks. I sometimes also use something I picked up from Lambert Meertens some 25 years ago: leave a specific mark in the current source file at the place of the cursor. The mark I use is "HIRO," in his honor. It is colloquial Dutch for "here" and selected for its unlikeliness to ever occur in finished code. :-)

At Google we also have tools integrated with Perforce that help me in an even earlier stage: when I come in to work, I might execute a command that lists each of the unfinished projects in my workspace, so as to remind me which projects I was working on the previous day. I also keep a diary in which I occasionally record specific hard-to-remember strings (like shell commands or URLs) that help me perform specific tasks for the project at hand—for example, the full URL to a server stats page, or the shell command that rebuilds the components I'm working on.

What are your suggestions to design an interface or an API?

Guido: Another area where I haven't spent a lot of conscious thought about the best process, even though I've designed tons of interfaces (or APIs). I wish I could just include a talk by Josh Bloch on the subject here; he talked about designing Java APIs, but most of

what he said would apply to any language. There's lots of basic advice like picking clear names (nouns for classes, verbs for methods), avoiding abbreviations, consistency in naming, providing a small set of simple methods that provide maximal flexibility when combined, and so on. He is big on keeping the argument lists short: two to three arguments is usually the maximum you can have without creating confusion about the order. The worst thing is having several consecutive arguments that all have the same type; an accidental swap can go unnoticed for a long time then.

I have a few personal pet peeves: first of all, and this is specific to dynamic languages, don't make the return type of a method depend on the *value* of one of the arguments; otherwise it may be hard to understand what's returned if you don't know the relationship—maybe the type-determining argument is passed in from a variable whose content you can't easily guess while reading the code.

Second, I dislike "flag" arguments that are intended to change the behavior of a method in some big way. With such APIs the flag is always a constant in actually observed parameter lists, and the call would be more readable if the API had separate methods: one for each flag value.

Another pet peeve is to avoid APIs that could create confusion about whether they return a new object or modify an object in place. This is the reason why in Python the list method `sort()` doesn't return a value: this emphasizes that it modifies the list in place. As an alternative, there is the built-in `sorted()` function, which returns a new, sorted list.

Should application programmers adopt the "less is more" philosophy? How should they simplify the user interface to provide a shorter learning path?

Guido: When it comes to graphical user interfaces, it seems there's finally growing support for my "less is more" position. The Mozilla foundation has hired Aza Raskin, son of the late Jef Raskin (codesigner of the original Macintosh UI) as a UI designer. Firefox 3 has at least one example of a UI that offers a lot of power without requiring buttons, configuration, preferences or anything: the smart location bar watches what I type, compares it to things I've browsed to before, and makes useful suggestions. If I ignore the suggestions it will try to interpret what I type as a URL or, if that fails, as a Google query. Now that's smart! And it replaces three or four pieces of functionality that would otherwise require separate buttons or menu items.

This reflects what Jef and Aza have been saying for so many years: the keyboard is such a powerful input device, let's use it in novel ways instead of forcing users to do everything with the mouse, the slowest of all input devices. The beauty is that it doesn't require new hardware, unlike Sci-Fi solutions proposed by others like virtual reality helmets or eye movement sensors, not to mention brainwave detectors.

There's a lot to do of course—for example, Firefox's Preferences dialog has the dreadful look and feel of anything coming out of Microsoft, with at least two levels of tabs and many modal dialogs hidden in obscure places. How am I supposed to remember that in order to turn off JavaScript I have to go to the Content tab? Are Cookies under the Privacy

tab or under Security? Maybe Firefox 4 can replace the Preferences dialog with a "smart" feature that lets you type keywords so that if I start typing "pass," it will take me to the section to configure passwords.

What do the lessons about the invention, further development, and adoption of your language say to people developing computer systems today and in the forseeable future?

Guido: I have one or two small thoughts about this. I'm not the philosophical kind, so this is not the kind of question I like or to which I have a prepared response, but here's one thing I realized early on that I did right with Python (and which Python's predecessor, ABC, didn't do, to its detriment). A system should be extensible by its users. Moreover, a large system should be extensible at two (or more) levels.

Since the first time I released Python to the general public, I got requests to modify the language to support certain kinds of use cases. My first response to such requests is always to suggest writing some Python code to cover their needs and put it in a module for their own use. This is the first level of extensibility—if the functionality is useful enough, it may end up in the standard library.

The second level of extensibility is to write an extension module in C (or in C++, or other languages). Extension modules can do certain things that are not feasible in pure Python (though the capabilities of pure Python have increased over the years). I would much rather add a C-level API so that extension modules can muck around in Python's internal data structures, than change the language itself, since language changes are held to the highest possible standard of compatibility, quality, semantic clarity, etc. Also, "forks" in the language might happen when people "help themselves" by changing the language implementation in their own copy of the interpreter, which they may distribute to others as well. Such forks cause all sorts of problems, such as maintenance of the private changes as the core language also evolves, or merging multiple independently forked versions that other users might need to combine. Extension modules don't have these problems; in practice most functionality needed by extensions is already available in the C API, so changes to the C API are rarely necessary in order to enable a particular extension.

Another thought is to accept that you don't get everything right the first time. Early on during development, when you have a small number of early adopters as users, is the time to fix things drastically as soon as you notice a problem, never mind backward compatibility. A great anecdote I often like to quote, and which has been confirmed as truthful by someone who was there at the time, is that Stuart Feldman, the original author of "Make" in Unix v7, was asked to change the dependence of the Makefile syntax on hard tab characters. His response was something along the lines that he agreed tab was a problem, but that it was too late to fix since there were already a dozen or so users.

As the user base grows, you need to be more conservative, and at some point absolute backward compatibility is a necessity. There comes a point where you have accumulated so many misfeatures that this is no longer feasible. A good strategy to deal with this is

what I'm doing with Python 3.0: announce a break with backward compatibility for one particular version, use the opportunity to fix as many such issues as possible, and give the user community a lot of time to deal with the transition.

In Python's case, we're planning to support Python 2.6 and 3.0 alongside each other for a long time—much longer than the usual support lifetime of older releases. We're also offering several transitional strategies: an automated source-to-source conversion tool that is far from perfect, combined with optional warnings in version 2.6 about the use of functionality that will change in 3.0 (especially if the conversion tool cannot properly recognize the situation), as well as selective back-porting of certain 3.0 features to 2.6. At the same time, we're not making 3.0 a total rewrite or a total redesign (unlike Perl 6 or, in the Python world, Zope 3), thereby minimizing the risk of accidentally dropping essential functionality.

One trend I've noticed in the past four or five years is much greater corporate adoption of dynamic languages. First PHP, Ruby in some context, definitely Python in other contexts, especially Google. That's interesting to me. I wonder where these people were 20 years ago when languages like Tcl and Perl, and Python a little bit later, were doing all of these useful things. Have you seen desire to make these languages more enterprise-friendly, whatever that means?

Guido: Enterprise-friendly is usually when the really smart people lose interest and the people of more mediocre skills have to somehow fend for themselves. I don't know if Python is harder to use for mediocre people. In a sense you would think that there is quite a bit of damage you cannot do in Python because it's all interpreted. On the other hand, if you write something really huge and you don't use enough unit testing, you may have no idea what it actually does.

You've made the argument that a line of Python, a line of Ruby, a line of Perl, a line of PHP, may be 10 lines of Java code.

Guido: Often it is. I think that the adoption level in the enterprise world, even though there are certain packages of functionality that are helpful, is probably just a fear of very conservative managers. Imagine the people in charge of IT resources for 100,000 people in a company where IT is not a main product—maybe they are building cars, or doing insurance, or something else, but everything they do is touched by computers. The people in charge of that infrastructure necessarily have to be very conservative. They will go with stuff that looks like it has a big name attached, like maybe Sun or Microsoft, because they know that Sun and Microsoft screw up all the time, but these companies are obliged to recover from those screwups and fix them, even if it takes five years.

Open source projects traditionally have just not offered that same peace of mind to the average CIO. I don't know exactly if and how and when that will change. It's possible that if Microsoft or Sun suddenly supported Python on their respective VMs, programmers in enterprises would actually discover that they can get higher productivity without any downsides by using more advanced languages.

CHAPTER THREE

APL

In the late 1950s, while on the faculty of Harvard University, Kenneth Iverson devised an extension of mathematical notation for the precise description of algorithms. Then, along with Adin Falkoff and other researchers at IBM, the team gradually turned the notation into a full-fledged programming language called APL. The language uses an extended character set requiring a specialized keyboard and appears on the page as strings of sometimes unfamiliar symbols—but the underlying consistency of the language makes it easy to learn, and its unmatched array-processing capabilities make it extraordinarily powerful. Its spiritual descendents, J and K, continue APL's legacy of concise and powerful algebraic manipulations.

Paper and Pencil

I read a paper written by you and Ken Iverson, "The Design of APL," which said that the first seven or eight years of development happened without any computer involved! This let you change design aspects without having to worry about legacy issues. How did the first software implementation influence the evolution of the language?

Adin Falkoff: Yes, the first years of the evolution of APL, when it had no name other than "Iverson's notation," were mainly concerned with paper-and-pencil mathematical applications, analysis of digital systems, and teaching. To a great extent, we thought of programming as a branch of mathematics concerned with the discovery and design of algorithms, and this concept was supported by the symbolic form of the notation. The attractiveness of the notation as a general programming language became evident after a while, and was advanced by the efforts of various people (in particular, Herb Hellerman at IBM) who experimented with machine implementations of significant elements of the notation, including primitive functions and array operations. Nevertheless, it is true that throughout this period we had complete freedom to design the language without concern for "legacy" issues.

The most significant early evolution of the language took place in two steps. First was the writing and publication of "The Formal Description of System 360" [*IBM Systems Journal*, 1964]. In order to formally describe some of the behavior of this newly designed computing system, some additions and modifications to the notation described in Iverson's book (*A Programming Language* [Wiley]) were necessary. Second was the design of a type element for Selectric-based terminals, which we undertook in anticipation of using the language on a machine. This imposed significant restraints arising from the linear nature of typewriting, and mechanical requirements of the Selectric mechanism. I believe there is considerable detail on the influence of these two factors on the evolution of the language in the paper you refer to, "The Design of APL" [*IBM Journal of Research and Development*, 1974].

The first comprehensive implementation of the language was, of course, APL\360. It necessarily introduced facilities to write defined functions (i.e., programs)—something taken for granted when using pencil and paper—and for controlling the environment in which programs would be executed. The ideas introduced then, including the workspace and library system, rules for scope of names, and the use of shared variables for communication with other systems, have persisted without significant change. Programs written for APL\360 run without modification on the modern APL systems that I am familiar with.

It is fair to say that the presence of an implementation influenced further evolution of the language by the strict application of the principle that new ideas must always subsume the earlier ones, and, of course, by the constant critical examination of how the language was working for new and different applications.

When you defined the syntax, how did you picture the typical APL programmer?

Adin: We did not direct our thinking about syntax to programmers as such, but rather conceived the language as being a communication medium for people, which incidentally should also work for people communicating with machines. We did realize that users would have to be comfortable with a symbolic language like algebra, but also felt that they would come to appreciate the power of symbolic representation, as it facilitates formal manipulation of expressions leading to more effective analysis and synthesis of algorithms. Specifically, we did not believe a lot of experience or knowledge of mathematics was necessary, and in fact used the APL system for teaching at the elementary and high school level with some notable success.

As time went on, we found that some of the most skilled and experienced programmers were attracted to APL, used it, and contributed to its development.

Did the complex syntax limit the diffusion of APL?

Adin: The syntax of APL and its effect on the acceptance of the language is well worth discussing, although I do not agree with the statement that it is "complex." APL was based on mathematical notation and algebraic expressions, regularized by removing anomalous forms and generalizing accepted notation. For example, it was decided that dyadic functions like addition or multiplication would stand between their two arguments, and monadic functions would consistently have the function symbols written before the argument, without exceptions such as are found in traditional math notation, so that absolute value in APL has one vertical bar before the argument and not bars on both sides, and the symbol for factorial in APL comes before the argument rather than following it. In this respect, the syntax of APL was simpler than the syntax of its historical source.

The syntax of APL was also simpler than that of algebraic notation and other programming languages in another very important way: the precedence rule for the evaluation of expressions in APL is simply that all functions have the same precedence, and the user does not have to remember whether exponentiation is carried out before multiplication, or where defined functions fit into the hierarchy. The rule is simply that the rightmost subexpression is evaluated first.

Hence, I don't believe that the syntax of APL limited the diffusion of the language, although the character set, using many nonalphabetic symbols not easily available on standard keyboards, probably did have such an effect.

How did you decide to use a special character set? How did that character set evolve over time?

Adin: The character set was defined by the use of conventional mathematical notation, augmented by a few Greek letters and some visually suggestive symbols like the quad.

There was also the practical influence of the linear typewriter limitation, leading to the invention of some characters that could be produced by overstriking. Later on, as terminals and input devices became more versatile, these composite characters became primitive symbols in their own right, and a few new characters were introduced to accommodate new facilities, such as the diamond for a statement separator.

Was there a conscious decision to use the limited resources of the time more productively?

Adin: The character set definitely was influenced by the desire to optimize the use of the limited resources available at the time; but the concise, symbolic form was developed and maintained because of the conviction that it facilitated analysis and formal manipulation of expressions. Also, the brevity of programs compared to equivalent ones written in other languages makes it easier to comprehend the logical flow of a program once the effort is made to read it in the concise APL representation.

I would think people needed a lot of training to learn the language, especially the character set. Was there a process of natural selection, which meant that APL programmers were experts at the language? Were they more productive? Did they write higher-quality code with fewer bugs?

Adin: Learning APL to the point of being able to write programs at the level of FORTRAN, for example, was actually not difficult or lengthy. Programming in APL was more productive because of the simplicity of the rules, and the availability of primitive functions for data manipulation like sorting, or mathematical functions like matrix inversion. These factors contributed to the conciseness of APL programs, which made them easier to analyze and debug. Credit for productivity must also be given to the APL implementations, using workspaces with all their useful properties, and the interactive terminal-based interpretive systems.

A super-concise form of expression might be incredibly useful on devices with a small screen like PDAs or smartphones! Considering that APL was first coded on big iron such as IBM System/360, would it be extensible to handle modern projects that need to manage network connections and multimedia data?

Adin: An implementation of APL on a handheld device would at the very least provide a very powerful hand calculator; and I see no problem with networks and multimedia, as such applications have been managed in APL systems for a very long time. Tools for managing GUIs are generally available on modern APL systems.

Early on in the development of APL systems, facilities for managing host operating systems and hardware from within APL functions were introduced, and were utilized by APL system programmers to manage the performance of APL itself. And commercial APL timesharing systems dependent upon networks for their economic viability used APL for managing their networks.

It is true that the first commercially viable APL systems were coded on large machines, but the earliest implementations, which demonstrated the feasibility of APL systems, were done on relative small machines, such as the IBM 1620 and the IBM 1130 family, including the IBM 1500, which had significant usage in educational applications. There was even an implementation on an early experimental desktop machine, dubbed "LC" for "low cost," that had but a few bytes of memory and a low-capacity disk. The evolution of IBM APL implementation is described in some detail in the paper "The IBM Family of APL Systems" [*IBM Systems Journal*, 1991].

Elementary Principles

When you pursued standardization, was it a deliberate decision?

Adin: We surely started standardization fairly early; in fact I think I wrote a paper about it, and we got to be part of ISO. We always wanted to standardize things and we managed to a large extent to do that. We discouraged people from fiddling around with the basic structures of the language, adding arbitrary kind of things that would complicate the syntax, or violate some of the elementary principles we were trying to maintain.

What was your main desire for standardization, compatibility or conceptual purity?

Adin: The desire of standardization is an economic issue. We surely wanted APL to be viable economically, and since a lot of different people were implementing and using it, it seemed a good idea to have a standard.

Several different vendors had different APL compilers. Without strong standardization, what happens when you have an extension that works on one system but not on another?

Adin: That is something worked on rather carefully by the APL standardization committees, and efforts were made to compromise between extensibility and purity.

You want people to be able to solve problems you haven't anticipated, but you don't want them to remove the essential nature of your system. Forty years later, how do you think the language holds up? Are the design principles you chose still applicable?

Adin: I think so; I really don't see anything really wrong.

Is that because you spent a lot of time designing it carefully or because you had a very strong theoretical background with algebra?

Adin: I think we were a couple of reasonably smart people with a belief in the concepts of simplicity and practicality, and an unwillingness to compromise that vision.

I found it too much trouble to try to learn and remember all the rules in other languages so I tried to keep it simple from that standpoint, so that I could use it.

Some of our way of thinking shows up in papers, especially the ones jointly authored by Iverson and me. I myself later wrote a paper that was called "A Note on Pattern Matching: Where do you find the match to an empty array?"[APL Quote Quad, 1979], which used some nice reasoning involving small programs and algebraic principles, to obtain the reported results, which turned out to be consistent and useful. The paper looked at various possibilities, and found that the one simplest to express works out better than any other.

I found it really fascinating to build a language from a small set of principles and discovering new ideas built on those principles. That seems like a good description of mathematics. What is the role of math in computer science and programming?

Adin: I believe that computer science is a branch of mathematics.

Programming of mathematical computations is obviously part of mathematics, especially the numerical analysis required to constantly maintain compatibility between discrete digital operations and the continuity of theoretical analysis.

Some other thoughts that come to mind are: the impetus from math problems that can be solved only by extensive computations that inspire need for speed; the discipline of logical thought required for math and carried over to programming of all kinds; the notion of algorithms, which are a classical mathematical tool; and the various specialized branches of mathematics, such as topology, that lend themselves to analysis of computational problems.

I have read some other discussions where you and other people suggested that one of the interesting applications was using APL to teach programming and mathematics at the elementary and high school levels.

Adin: We did some of that, particularly at the beginning, and we had a little fun with it.

At that time we only had typewriter terminals and we made some available to some local private schools. There was one in particular where problem students were supposed to be taught, and we gave them exercises to do on the typewriter and turned them loose.

The fun part was that we found that some of these students who were supposed to be resistant to learning broke into the school after hours so they could do more work on it. They were using typewriter terminals hooked to our time-sharing system.

So they enjoyed that so much they suddenly had to do it even afterward?

Adin: Yes.

You used APL to teach "programming thinking" to nonprogrammers. What made APL attractive for nonprogrammers?

Adin: In the early days one of the things was you didn't have all this overhead, you didn't have to make declarations before you added two numbers, so if you wanted to add 7 and 5 you just wrote down 7 + 5, instead of saying there is a number called 7 and there is a number called 5, these are numbers, floating point or not floating point, and the result is a number and I want to store the result here, so there was a lower barrier in APL to doing what you wanted.

When someone is learning to program, the initial step toward doing that first thing is very small. You basically write down what you want to do, and you don't have to spend time pleasing a compiler to get it to work.

Adin: That's right.

Easy to start and easy to play with. Does this technique let people become programmers or increase their programming knowledge?

Adin: The easy accessibility makes it easy to experiment, and if you can experiment and try out different things, you learn, and so I think that is favorable toward the development of programming skills.

The notation that you chose for APL is different from traditional algebraic notation.

Adin: Well, it's not that different…the precedence rules are different. They are very simple: you go from right to left.

Did you find that much easier to teach?

Adin: Yes, because there is only one rule and you don't have to say that if it's a defined function, you go this way, and if it's exponentiation, it has precedence over multiplication, or stuff like that. You just say, "look at the line of the instructions and take it from right to left."

Was this a deliberate design decision to break with familiar notation and precedence in favor of greater simplicity?

Adin: That's right. Greater simplicity and greater generality.

I think Iverson was mainly responsible for that. He was quite good at algebra and he was very interested in teaching. One example he liked to use was the representation of polynomials, which is extremely simple in APL.

When I first saw that notation, even though it was unfamiliar, it did seem conceptually much simpler overall. How do you recognize simplicity in a design or an implementation? Is that a matter of good taste or experience, or is there a rigorous process you apply to try to find optimal simplicity?

Adin: I think to some extent it must be subjective, because it depends somewhat on your experience and where you come from. I would say the fewer there rules are, the simpler it is in general.

You started from a small set of axioms and you can build from there, but if you understand that small set of axioms, you can derive more complexity?

Adin: Well, let's take this matter of precedence. I think it's simpler to have the precedence based on a simple form from right to left, than on a basis of a table that says this function goes first and that function goes second. I think it is one rule versus an almost limitless number of rules.

You see, in any particular application you set up your own set of variables and functions, and for a particular application you might find it simpler to write some new rules, but if you are looking at a general language like APL, you want to start with the fewest possible number of rules.

To give people designing systems built with the language more opportunity to evolve?

Adin: People who are building applications are in fact building languages; fundamentally, programming has to do with developing languages suitable for particular applications.

You express the problem in a language specific to its domain.

Adin: But then those objects, notably the nouns and the verbs, the objects and the functions, they have to be defined in something, for example in a general-purpose language like APL.

So you use APL to define these things, but then you set up your operations to facilitate the kind of things you want to do in that application.

Is your concern constructing the building blocks people can use to express themselves?

Adin: My concern is giving them the basic building blocks if you like, the fundamental tools for constructing the building blocks that are suitable and appropriate for what they are trying to accomplish in the field in which they are working.

It seems to be a concern shared by other language designers; I think of Chuck Moore with Forth, or John McCarthy with Lisp, and Smalltalk in the early 70s.

Adin: I'm sure that's the case.

McCarthy, I know, is a theoretical kind of person and he was concerned with developing a system to express the lambda calculus effectively, but I don't think the lambda calculus is as convenient for most purposes as plain old algebra, from which APL derives.

Suppose I want to design a new programming language. What's the best piece of advice you can give me?

Adin: I guess the best thing I can say is do something that you enjoy, something that pleases you to work with, something that helps you accomplish something that you would like to do.

We were always very personal in our approach, and I think most designers are, as I read what people have to say. They started doing things that they wanted to do, which then turned out to be useful generally.

When you were designing APL, were you able to see at some point "we are going in the wrong direction here; we need to scale back this complexity" or "we have several different solutions; we can unify them into something much simpler"?

Adin: That is approximately right, but there was usually a question of "is this a generalization which subsumes what we already have, and what is the likelihood that it is going to enable us to do a lot more with very little further complication?"

We paid a lot of attention to end conditions—what happens in a limit when you go from 6 to 5 to 4 down to 0, for example. Thus, in reduction you are applying a function like summation to a vector, and if you are summing up a vector that has n elements and then n minus one elements, and so on, what happens when you eventually have no elements? What's the sum? It has to be 0 because that's the identity element.

In the case of multiplication, the multiplication over an empty vector goes to 1, because that's the identity element for that function.

You mentioned looking at several different solutions and trying to generalize and asking yourself the question of what happens when approaching 0, for example. If you hadn't already known that when you do a reduction, you need to end up at the identity element for when n *is 0, you could look at both those cases and say "Here is the argument we make: it is 0 when this case and it's 1 in this case, because it is the identity element."*

Adin: That's right. That's one of the processes we used.

What happens in the special cases is very important, and when you use APL effectively, you keep applying that criterion to the more elaborate functions that you might be developing for a particular application. This often leads to unexpected but gratifying simplification.

Do the design techniques you use when creating a language inform the design techniques people might use when programming in the language?

Adin: Yes, because as I said before, programming is a process of designing languages. I think that's a very fundamental thing, which is not often mentioned in the literature as far as I know.

Lisp programmers do, but in a lot of the languages that came afterward, especially Algol and its C derivatives, people don't seem to think this way. Is there a divide between what is built in the language and what's not, where everything else is second class?

Adin: Well, what do we mean by second class? In APL the so-called second class follows the same rules as the first class, and we don't have any problem there.

You can make the same argument for almost all of Lisp or Scheme or Smalltalk, but C has a distinct division between operators and functions, and user-created functions. Is making that distinction sharp between these entities a design mistake?

Adin: I don't know if I would call it a mistake, but I think it's simpler to have the same rules apply to both what's primitive and not primitive.

What's the biggest mistake you've made with regard to design or programming? What did you learn from it?

Adin: When work on APL first began, we consciously avoided making design decisions that catered to the computer environment. For example, we eschewed the use of declarations, seeing their use as an unnecessary burden on the user when the machine could easily determine the size and type of a data object from the object itself at the time of its input or generation. In the course of time, however, as APL became more widely used with more and more vested interests, hardware factors were increasingly difficult to avoid.

Perhaps the biggest mistake that I personally made was to underestimate advances in hardware and become too conservative in system design. In contemplating early implementation of APL on the PC, for instance, I advocated leaving out recent language extensions to general arrays and complex numbers because these would strain the capacity of the extant hardware to provide satisfactory performance. Fortunately, I was overruled, and it was not long before major increases in PC memory and processor speeds made such powerful extensions completely feasible.

It is hard to think of big mistakes made in programming because one expects to make errors in the course of writing a program of reasonable complexity. It then depends on the programming tools how the error grows, when it is discovered, and how much has to be redone to recover from it. Modularization and ready reuse of idiomatic code fragments, as follows from the functional programming style fostered by APL, tends to limit the generation and propagation of errors so they don't become big mistakes.

As for mistakes in the design of APL itself, our method of development, using consensus among the designers and implementers as the ultimate deciding factor, and feedback from users gaining practical experience in a diversity of applications as well as our own use of the language before design was frozen, helped us avoid serious errors.

However, one person's exercise of principle may be another's idea of a mistake, and even over long periods of time, differences may not be empirically resolvable. Two things come to mind.

One is the character set. There was from the earliest times considerable pressure to use reserved words instead of the abstract symbols chosen to represent primitive functions. Our position was that we were really dealing with extensions to mathematics, and the evolution of mathematical notation was clearly in the direction of using symbols, which facilitated formal manipulation of expressions. Later on, Ken Iverson, who had an abiding interest in the teaching of mathematics, chose to limit the character set to ASCII in his further work, on the language J, so that J systems could be easily accessible to students and others without specialized hardware. My own inclination was and is to stick with the symbolic approach; it's more in keeping with history and ultimately easier to read. Time will tell if either direction is mistaken, or if it doesn't really matter.

The second thing that comes to mind as possibly leading to a significant mistake in direction that may never be decided is the treatment of general arrays, i.e., arrays whose scalar

elements may themselves have an accessible structure within the language. After APL\360 was established as an IBM product (one of the very first such when IBM unbundled its software and hardware in 1966 or 1967), we began to look at extensions to more general arrays and had extensive studies and discussions regarding the theoretical underpinnings. Ultimately APL systems have been built with rival ways of treating scalar elements and syntactic consequences. It will be interesting to see how this evolves as the general interest in parallel programming becomes more commercially important.

Parallelism

What are the implications (for the design of applications) of thinking about data in collections rather than as individual units?

Adin: This is a rather large subject, as indicated by the spread of "array languages" and the introduction of array primitives in languages like FORTRAN, but I think there are two significant aspects to thinking in terms of collections.

One, of course, is the simplification of the thought process when not bogged down in the housekeeping details of dealing with individual items. It is closer to our natural way of thinking to say, for example, how many of the numbers in this collection are equal to zero, and write a simple expression that produces the desired result, than to start thinking in terms of a loop in any of its derivative forms.

The second is that possibilities for parallelism are made more evident in programs acting directly on collections, leading to more efficient utilization of modern hardware.

There's been some talk in modern programming languages about adding higher-order features to languages such as C++ or Java—languages where you spend a lot of time writing the same `for( )` *loop over and over again. For example, I have a collection of things and I want to do something to each one of them. Yet APL solved this problem 40–45 years ago!*

Adin: Well, I don't know how many years ago, but there are sort of two stages there. One is the use of arrays as primitive, and second stage was the introduction of the operator called `each`, which basically applies any arbitrary function to any collection of items. But there were always some questions like "Do we want to put in primitives for looping specifically?" We decided we didn't want to do that because it complicated the syntax too much, and it was easy enough to write the few needed loops in the standard way.

Complicate the syntax for the implementation or for users?

Adin: For both: people have to read it, machines have to read it; the syntax is either simple or not.

You would put in new kinds of statements, and that's clearly a complication. Now the question is "Is the payoff worth it?", and that's where the design judgment comes in. And we always came down on the side that we didn't want to have new kinds of syntax for handling loops since we could do it quite conveniently with what we had.

You said that APL really has an advantage for parallel programming. I can understand the use of arrays as the primitive data structure for the language. You also mentioned the use of shared variables. How do they work?

Adin: A shared variable in APL is a variable that is accessible to more than one processor at a time. The sharing processors can both be APL processors or one can be of a different sort. For example, you can have a variable, let's call it X, and, as far as APL is concerned, reading and writing X is not different from an ordinary variable. However there might be another processor, let's say a file processor, which also has access to X, it being a shared variable, and whatever value APL might give to X, the file processor uses that value according to its own interpretation. And similarly, when it gives a value to X, which is then read by APL, the APL processor similarly applies its own knowledge to it, however it chooses to interpret that value. And this X is a shared variable.

What we have in APL systems like APL2 of IBM is some protocol for managing access to this variable so that you don't run into trouble with different kinds of race conditions.

Is this parallelization you were talking about something the compiler can determine automatically? Suppose that I want to multiply two arrays and add the value to each element of an array. This is easy to express in APL, but can the compiler perform implicit parallelization on that?

Adin: The definition in APL is that it doesn't matter in what order you do the operations on the elements of an array; therefore, the compiler or the interpreter or whatever implementation you have is free to do them simultaneously or in any arbitrary sequence.

Besides enabling simplicity at the language level, it can give implementers tremendous flexibility to change the way the implementation works, taking advantage of new hardware, or give you a mechanism to exploit things like automatic parallelization.

Adin: That's right, because according to the definition of the language, which is of course the definition of what happens when the processor is applied, it doesn't matter what order you do them. That was a very deliberate decision.

Was that a unique decision in the history of languages of the time?

Adin: I am not that familiar with the history of languages, but since we were basically the only serious array-oriented language, it probably was unique.

It's interesting to talk about collections and large data sets, which are clearly preoccupations of modern programmers. APL preceded the invention of the relational database. Now we have a lot of data in structures containing different data types, in relational databases, and in large unstructured collections such as web pages. Can APL handle these well? Does it offer models that people using more popular languages such as SQL, PHP, Ruby, and Java can learn from?

Adin: APL arrays can have as elements both scalars, which have no internal structure, and nonscalars, which may be of any complexity. Nonscalar elements are recursively structured of other arrays. "Unstructured" collections such as web pages can therefore be conveniently represented by APL arrays and manipulated by primitive APL functions.

Regarding very large arrays, APL has the facility to treat external files as APL objects. Once an association has been made between a name in the workspace and an external file, operations can be applied to the file using APL expressions. It appears to the user as if the file is within the workspace, even though in actuality it may be many times larger than the workspace size.

It is very hard to give specific details of what designers of other languages can learn from APL, and it would be presumptuous of me to go into particulars of the languages you mention, as I am not an expert in any of them. However, as I read about them in the literature I see that by and large the principles that guided the design of APL—which we described, for example, in our 1973 paper "The Design of APL"—have continued to inform later work in language design.

Of the two overriding principles, simplicity and practicality, the latter seems to have fared better; simplicity is a more difficult objective to achieve since there are no practical constraints on complexity. We strove for simplicity in APL by carefully defining the scope of the primitive operations it would allow, maintaining the abstract nature of APL objects, and resisting the temptation to include special cases represented by the operations of other systems.

An illustration of this is the fact that the concept of a "file" does not appear in APL. We have arrays that may be treated as files as called for by an application, but there are no primitive functions specifically designed for file manipulation as such. The practical need for efficiency in file management, however, early on fostered the development of the shared-variable paradigm, which itself is a general concept useful in a multitude of applications where the APL program needs to invoke facilities of another (APL or non-APL) auxiliary processor.

Later on, an additional facility, using the general concept of namespaces, was designed to allow APL programs to directly manipulate objects outside of the workspace, including access to Java fields and methods, extremely large data collections, compiled programs in other languages, and others. The user interface to both the shared-variable and namespace facilities rigorously maintains APL syntax and semantics and thereby keeps it simple.

Without going into detail, therefore, it is reasonable to say that the newer languages could benefit by maintaining a strict adherence to their own primitive concepts, defining each to be as general as possible within the context of the applications they are addressing.

As for specific characteristics of APL as a model, APL has demonstrated that declarations are unnecessary, although they may contribute to efficiency of execution in some situations, and that the number of different data types can be quite small. Newer languages may benefit by aiming in these directions rather than taking it for granted that the user has to help out the computer by providing such implementation-related information.

Also, the concept of a pointer is not a primitive in APL, and has never been missed. Of course, where possible the primitive operations in the language should be defined on collections of data having an abstract internal structure, such as regular arrays, trees, and others.

You are correct in noting that APL preceded the invention of the relational database. Both Dr. E. F. (Ted) Codd and the APL group were at the IBM T. J. Watson Research Center in the 1960s, when he was developing the relational database concepts, and I believe that we had a very strong influence on that work. I recall in particular a heated discussion between us one afternoon where we demonstrated that simple matrices, rather than complex scalar pointer systems, could be used for representing the relationships among data entities.

Legacy

I know that lots of design influences in Perl came from APL. Some people say some of the crypticness of Perl comes from APL. I don't know if this is a compliment or not.

Adin: Let me give you an example of that kind of compliment. There is a lot of politics involved in the design and use of programming languages, particularly in a place like IBM where it is a business. At various times, people tried to set up competitive experiments to see if APL would do better than, say, PL1 or FORTRAN. The results were always loaded, because the judges were people on the other side, but there is one comment that I always remember from some functionary: he said APL can't be very good because two of the smartest guys he knew, Iverson and Falkoff, can't make people believe in it.

What do the lessons about the invention, further development, and adoption of your language say to people developing computer systems today and in the foreseeable future?

Adin: Decisions about system design are not purely technical or scientific. Economic and political considerations have a strong influence, and especially so in situations where there is potential flexibility in the underlying technicalities, as in the design of languages and systems.

In the period when APL was taking hold as an important tool being used within IBM in the mid-1960s, and consideration was being given to making it into a product, we had to contend with an IBM "language czar," who decreed that only PL/1 would be supported by the company in the future—except, of course, for FORTRAN and COBOL, which were already entrenched in the industry and could not be totally abandoned.

As history has shown, this was an unrealistic position for the company to take and was bound to fail, but this was not so obvious at the time, considering the dominance of IBM in the computing industry and the dominance of certain factions within the power structure of the company.

We had to fight the policy to get the necessary support for APL to survive. The battle took place on several fronts: as members of the IBM Research Division, we exploited as much as possible opportunities to give professional talks, seminars, and formal classes so as to imbed awareness of APL's unique characteristics in the technical consciousness of the time; we enlisted—wherever we could find them—people of influence within the company to countervail against the administrative power structure; we spread and supported

the internal use of our APL\360 system to development and manufacturing locations; we leveraged important customers' interest in APL systems to force the availability of APL outside the company, at least on an experimental basis; and made allies within the ranks of the marketing division. And we were successful, to the point where APL\360 was among the very first IBM program products to be marketed after the unbundling of hardware and software in the late 1960s.

A very significant milestone was accomplished on account of the interest that technical talks and demonstrations had engendered at the NASA Goddard Space Center. In 1966 that facility requested access to our internal APL system in order to experiment with its use. They were a very important customer, and we were urged by the IBM marketing people to comply with their request. However, we demurred, insisting that we would only agree to do this if we were first enabled to give a weeklong instructional course on site at the Goddard Space Center.

We obtained this agreement, but then ran into difficulty implementing it: time-sharing systems like APL\360 at the time required terminals connecting to the central system through acoustic modems working with specialized telephone "data sets." These telephone sets were also used on the other end, attached to the central computer, and they were in short supply. After all the administrative agreements to go ahead with the project had been reached, we found that neither the New York- nor Washington D.C.-area phone companies could provide the units needed for the projected classes at the Space Center.

While it was their normal practice to work only with their own equipment, the D.C. phone company agreed to install any data sets we could somehow provide. But as much as our IBM communication managers tried to persuade the New York phone company to find data sets somewhere, they were not able to produce any, although they somehow conveyed the idea that they would look the other way if we happened to use their equipment already in our possession in ways they could not officially condone.

So we proceeded to disable half of the lines coming into our central computer, and had the data sets thus freed taken down to the Space Center in an IBM station wagon. They were then installed off the record by the local phone company and we were able to go ahead with our course, thus establishing the first off-premises use of the APL\360 system by a non-IBM entity, getting it out the door despite the support-only-PL/1 policy.

What do you regret most about the language?

Adin: We gave the design of APL our best efforts and worked hard in the political arena to have it accepted and widely used. Under the circumstances, I don't find anything to regret about the language. One possible regret in hindsight is that we did not start sooner and put greater effort behind the development of an effective compiler, but we can't know what this might have cost in tradeoffs, given the extant limitations of resources. Furthermore, there is reason to believe that current interest in parallel programming and the adoption of APL-like array operations in traditional compiled languages like FORTRAN will result in the equivalent in due course.

How do you define success in terms of your work?

Adin: APL proved to be a very useful tool in the development of many aspects of IBM's business. It provided a much simplified approach to using computers that allowed researchers and product developers to apply themselves more efficiently to the substantive problems they were working on, from theoretical physics to development of flat-screen displays. It was also used to prototype major business systems such as assembly lines and warehouses, allowing them to get started quickly and tested before being frozen in implementations using other programming systems.

We were successful in making APL into a whole line of IBM products, and providing leadership for other computer companies to provide their own APL systems conforming to an international standard.

APL also found substantial use in academic institutions as a tool and a discipline, thus fulfilling one of the principal purposes of its development—its use in education.

APL of course was the forerunner of programming languages and systems treating arrays as primitive data objects and using shared variables for managing simultaneity, and as such will no doubt have a strong influence on further developments involving parallel programming. It is very gratifying to see that in the last few months, three separate computer industry consortiums have been established to work in this field.

CHAPTER FOUR

Forth

Forth is a stack-based, concatenative language designed by Chuck Moore in the 1960s. Its main features are the use of a stack to hold data, and words that operate on the stack, popping arguments and pushing results. The language itself is small enough that it runs on anything from embedded machines to supercomputers, and expressive enough to build useful programs out of a few hundred words. Successors include Chuck Moore's own colorForth, as well as the Factor programming language.

The Forth Language and Language Design

How do you define Forth?

Chuck Moore: Forth is a computer language with minimal syntax. It features an explicit parameter stack that permits efficient subroutine calls. This leads to postfix expressions (operators follow their arguments) and encourages a highly factored style of programming with many short routines sharing parameters on the stack.

I read that the name Forth stands for fourth-generation software. Would you like to tell us more about it?

Chuck: Forth is derived from "fourth," which alludes to "fourth-generation computer language." As I recall, I skipped a generation. FORTRAN/COBOL were first-generation languages; Algol/Lisp, second. These languages all emphasized syntax. The more elaborate the syntax, the more error checking is possible. Yet most errors occur in the syntax. I determined to minimize syntax in favor of semantics. And indeed, Forth words are loaded with meaning.

You consider Forth a language toolkit. I can understand that view, given its relatively simple syntax compared to other languages and the ability to build a vocabulary from smaller words. Am I missing anything else?

Chuck: No, it's basically the fact that it's extremely factored. A Forth program consists of lots of small words, whereas a C program consists of a smaller number of larger words.

By small word, I mean one with a definition typically one line long. The language can be built up by defining a new word in terms of previous words and you just build up that hierarchy until you have maybe a thousand words. The challenge there is 1) deciding which words are useful, and 2) remembering them all. The current application I'm working on has a thousand words in it. And I've got tools for searching for words, but you can only search for a word if you remember that it exists and pretty much how it's spelled.

Now, this leads to a different style of programming, and it takes some time for a programmer to get used to doing it that way. I've seen a lot of Forth programs that look very much like C programs transliterated into Forth, and that isn't the intent. The intent is to have a fresh start. The other interesting thing about this toolkit, words that you define this way are every bit as efficient or significant as words that are predefined in the kernel. There's no penalty for doing this.

Does the externally visible structure consisting of many small words derive from Forth's implementation?

Chuck: It's a result of our very efficient subroutine call sequences. There's no parameter passing because the language is stack-based. It's merely a subroutine call and return. The stack is exposed. The machine language is compiled. A switch to and from a subroutine is literally one call instruction and one return instruction. Plus you can always reach down into the equivalent of an assembly language. You can define a word that will execute

actual machine instructions instead of subroutine calls, so you can be as efficient as any other language, maybe more efficient than some.

You don't have the C calling overhead.

Chuck: Right. This gives the programmer a huge amount of flexibility. If you come up with a clever factoring of a problem, you can not only do it efficiently, you can make it extraordinarily readable.

On the other hand, if you do it badly, you can end up with code that no one else can read—code your manager can't understand, if managers can understand anything. And you can create a real mess. So it's a two-edged sword. You can do very well; you can do very badly.

What would you say (or what code would you show) to a developer who uses another programming language to make him interested in Forth?

Chuck: It is very hard to interest an experienced programmer in Forth. That's because he has invested in learning the tools for his language/operating system and has built a library appropriate for his applications. Telling him that Forth would be smaller, faster, and easier is not persuasive compared to having to recode everything. A novice programmer, or an engineer needing to write code, doesn't face that obstacle and is much more receptive—as might be the experienced programmer starting a new project with new constraints, as would be the case with my multicore chips.

You mentioned that a lot of Forth programs you've seen look like C programs. How do you design a better Forth program?

Chuck: Bottom-up.

First, you presumably have some I/O signals that you have to generate, so you generate them. Then you write some code that controls the generation of those signals. Then you work your way up until finally you have the highest-level word, and you call it go and you type `go` and everything happens.

I have very little faith in systems analysts who work top-down. They decide what the problem is and then they factor it in such a way that it can be very difficult to implement.

Domain-driven design suggests describing business logic in terms of the customer's vocabulary. Is there a connection between building up a vocabulary of words and using the terms of art from your problem domain?

Chuck: Hopefully the programmer knows the domain before he starts writing. I would talk to the customer. I would listen to the words he uses and I would try to use those words so that he can understand what the program's doing. Forth lends itself to this kind of readability because it has postfix notation.

If I was doing a financial application, I'd probably have a word called "percent." And you could say something like "2.03 percent". And the argument's percent is 2.03 and everything works and reads very naturally.

How can a project started on punch cards still be useful on modern computers in the Internet era? Forth was designed on/for the IBM 1130 in 1968. That it is the language of choice for parallel processing in 2007 is surely amazing.

Chuck: It has evolved in the meantime. But Forth is the simplest possible computer language. It places no restrictions upon the programmer. He/she can define words that succinctly capture aspects of a problem in a lean, hierarchical manner.

Do you consider English readability as a goal when you design programs?

Chuck: At the very highest level, yes, but English is not a good language for description or functionality. It wasn't designed for that, but English does have the same characteristic as Forth in the sense that you can define new words.

You define new words by explaining what they are in previously defined words mostly. In a natural language, this can be problematic. If you go to a dictionary and check that out, you find that often the definitions are circular and you don't get any content.

Does the ability to focus on words instead of the braces and brackets syntax you might have in C make it easier to apply good taste to a Forth program?

Chuck: I would hope so. It takes a Forth programmer who cares about the appearance of things as opposed merely to the functionality. If you can achieve a sequence of words that flow together, it's a good feeling. That's really why I developed colorForth. I became annoyed at the syntax that was still present in Forth. For instance, you could limit a comment by having a left parenthesis and a right parenthesis.

I looked at all of those punctuation marks and said, "Hey, maybe there's a better way." The better way was fairly expensive in that every word in the source code had to have a tag attached to it, but once I swallowed that overhead, it became very pleasant that all of those funny little symbols went away and were replaced by the color of the word which was, to me, a much gentler way of indicating functionality.

I get interminable criticism from people who are color blind. They were really annoyed that I was trying to rule them out of being programmers, but somebody finally came up with a character set distinction instead of a color distinction, which is a pleasant way of doing it also.

The key is the four-bit tag in each word, which gives you 16 things that we're to do, and the compiler can determine immediately what's intended instead of having to infer it from context.

Second- and third-generation languages embraced minimalism, for example with meta-circular bootstrapping implementations. Forth is a great example of minimalism in terms of language concepts and the amount of hardware support required. Was this a feature of the times, or was it something you developed over time?

Chuck: No, that was a deliberate design goal to have as small a kernel as possible. Predefine as few words as necessary and then let the programmer add words as he sees fit.

The prime reason for that was portability. At the time, there were dozens of minicomputers and then there became dozens of microcomputers. And I personally had to put Forth on lots of them.

I wanted to make it as easy as possible. What happens really is there might be a kernel with 100 words or so that is just enough to generate a—I'll call it an operating system, but it's not quite—that has another couple hundred words. Then you're ready to do an application.

I would provide the first two stages and then let the application programmers do the third, and I was usually the application programmer, too. I defined the words I knew were going to be necessary. The first hundred words would be in machine language probably or assembler or at least be dealing directly with the particular platform. The second two or three hundred words would be high-level words, to minimize machine dependence in the lower, previously defined level. Then the application would be almost completely machine independent, and it was easy to port things from one minicomputer to another.

Were you able to port things easily above that second stage?

Chuck: Absolutely. I would have a text editor, for instance, that I used to edit the source code. It would usually just transfer over without any changes.

Is this the source of the rumor that every time you ran across a new machine, you immediately started to port Forth to it?

Chuck: Yes. In fact, it was the easiest path to understanding how the machine worked, what its special features were based on how easy it was to implement the standard package of Forth words.

How did you invent indirect-threaded code?

Chuck: Indirect-threaded code is a somewhat subtle concept. Each Forth word has an entry in a dictionary. In direct-threaded code, each entry points to code to be executed when that word is encountered. Indirect-threaded code points to a location that contains the address of that code. This allows information besides the address to be accessed—for instance, the value of a variable.

This was perhaps the most compact representation of words. It has been shown to be equivalent to both direct-threaded and subroutine-threaded code. Of course these concepts and terminology were unknown in 1970. But it seemed to me the most natural way to implement a wide variety of kinds of words.

How will Forth influence future computer systems?

Chuck: That has already happened. I've been working on microprocessors optimized for Forth for 25 years, most recently a multicore chip whose cores are Forth computers.

What does Forth provide? As a simple language, it allows a simple computer: 256 words of local memory; 2 push-down stacks; 32 instructions; asynchronous operation; easy communication with neighbors. Small and low-power.

Forth encourages highly factored programs. Such are well-suited to parallel processing, as required by a multicore chip. Many simple programs encourage thoughtful design of each. And requiring perhaps only 1% the code that would otherwise be written.

Whenever I hear people boasting of millions of lines of code, I know they have greviously misunderstood their problem. There are no contemporary problems requiring millions of lines of code. Instead there are careless programmers, bad managers, or impossible requirements for compatibility.

Using Forth to program many small computers is an excellent strategy. Other languages just don't have the modularity or flexibility. And as computers get smaller and networks of them are cooperating (smart dust?), this will be the environment of the future.

This sounds like one major idea of Unix: multiple programs, each doing just one thing, that interact. Is that still the best design today? Instead of multiple programs on one computer, might we have multiple programs across a network?

Chuck: The notion of multithreaded code, as implemented by Unix and other OSes, was a precursor to parallel processing. But there are important differences.

A large computer can afford the considerable overhead ordinarily required for multithreading. After all, a huge operating system already exists. But for parallel processing, almost always the more computers, the better.

With fixed resources, more computers mean smaller computers. And small computers cannot afford the overhead common to large ones.

Small computers will be networked, on chip, between chips and across RF links. A small computer has small memory. Nowhere is there room for an operating system. The computers must be autonomous, with a self-contained ability to communicate. So communication must be simple—no elaborate protocol. Software must be compact and efficient. An ideal application for Forth.

Those systems requiring millions of lines of code will become irrelevant. They are a consequence of large, central computers. Distributed computation needs a different approach.

A language designed to support bulky, syntactical code encourages programmers to write big programs. They tend to take satisfaction, and be rewarded, for such. There is no pressure to seek compactness.

Although the code generated by a syntactic language might be small, it usually isn't. To implement the generalities implied by the syntax leads to awkward, inefficient object code. This is unsuitable for a small computer. A well-designed language has a one-one correlation between source code and object code. It's obvious to the programmer what code will be generated from his source. This provides its own satisfaction, is efficient, and reduces the need for documentation.

Forth was designed partly to be compact in both source and binary output, and is popular among embedded developers for that reason, but programmers in many other domains have reasons to choose other languages. Are there aspects of the language design that add only overhead to the source or the output?

Chuck: Forth is indeed compact. One reason is that it has little syntax.

Other languages seem to have deliberately added syntax, which provides redundancy and offers opportunity for syntax checking and thus error detection.

Forth provides little opportunity for error detection due to its lack of redundancy. This contributes to more compact source code.

My experience with other languages has been that most errors are in the syntax. Designers seem to create opportunity for programmer error that can be detected by the compiler. This does not seem productive. It just adds to the hassle of writing correct code.

An example of this is type checking. Assigning types to various numbers allows errors to be detected. An unintended consequence is that programmers must work to convert types, and sometimes work to evade type checking in order to do what they want.

Another consequence of syntax is that it must accommodate all intended applications. This makes it more elaborate. Forth is an extensible language. The programmer can create structures that are just as efficient as those provided by the compiler. So all capabilities do not have to be anticipated and provided for.

A characteristic of Forth is its use of postfix operators. This simplifies the compiler and offers a one-one translation of source code to object code. The programmer's understanding of his code is enhanced and the resulting compiled code is more compact.

Proponents of many recent programming languages (notably Python and Ruby) cite readability as a key benefit. Is Forth easy to study and maintain in relation to those? What can Forth teach other programming languages in terms of readability?

Chuck: Computer languages all claim to be readable. They aren't. Perhaps it seems so to one who knows the language, but a novice is always bewildered.

The problem is the arcane, arbitrary, and cryptic syntax. All the parentheses, ampersands, etc. You try to learn why it's there and eventually conclude there's no good reason. But you still have to follow the rules.

And you can't speak the language. You'd have to pronounce the punctuation like Victor Borge.

Forth alleviates this problem by minimizing the syntax. Its cryptic symbols `@` and `!` are pronounced "fetch" and "store." They are symbols because they occur so frequently.

The programmer is encouraged to use natural-language words. These are strung together without punctuation. With good choice of words, you can construct reasonable sentences. In fact, poems have been written in Forth.

Another advantage is postfix notation. A phrase like "6 inches" can apply the operator "inches" to the parameter 6, in a very natural manner. Quite readable.

On the other hand, the programmer's job is to develop a vocabulary that describes the problem. This vocabulary can get to be quite large. A reader has to know it to find the program readable. And the programmer must work to define helpful words.

All in all, it takes effort to read a program. In any language.

How do you define success in terms of your work?

Chuck: An elegant solution.

One doesn't write programs in Forth. Forth is the program. One adds words to construct a vocabulary that addresses the problem. It is obvious when the right words have been defined, for then you can interactively solve whatever aspect of the problem is relevant.

For example, I might define words that describe a circuit. I'll want to add that circuit to a chip, display the layout, verify the design rules, run a simulation. The words that do these things form the application. If they are well chosen and provide a compact, efficient toolset, then I've been successful.

Where did you learn to write compilers? Was this something everybody at the time had to do?

Chuck: Well, I went to Stanford around '60, and there was a group of grad students writing an ALGOL compiler—a version for the Burroughs 5500. It was only three or four of them, I think, but I was impressed out of my mind that three or four guys could sit down and write a compiler.

I sort of said, "Well, if they can do it, I can do it," and I just did. It isn't that hard. There was a mystique about compilers at the time.

There still is.

Chuck: Yeah, but less so. You get these new languages that pop up from time to time, and I don't know if they're interpreted or compiled, but well, hacker-type people are willing to do it anyway.

The operating system is another concept that is curious. Operating systems are dauntingly complex and totally unnecessary. It's a brilliant thing that Bill Gates has done in selling the world on the notion of operating systems. It's probably the greatest con game the world has ever seen.

An operating system does absolutely nothing for you. As long as you had something—a subroutine called disk driver, a subroutine called some kind of communication support, in the modern world, it doesn't do anything else. In fact, Windows spends a lot of time with overlays and disk management all stuff like that which are irrelevant. You've got gigabyte disks; you've got megabyte RAMs. The world has changed in a way that renders the operating system unnecessary.

What about device support?

Chuck: You have a subroutine for each device. That's a library, not an operating system. Call the ones you need or load the ones you need.

How do you resume programming after a short hiatus?

Chuck: I don't find a short coding hiatus at all troublesome. I'm intensely focused on the problem and dream about it all night. I think that's a characteristic of Forth: full effort over a short period of time (days) to solve a problem. It helps that Forth applications are naturally factored into subprojects. Most Forth code is simple and easy to reread. When I do really tricky things, I comment them well. Good comments help re-enter a problem, but it's always necessary to read and understand the code.

What's the biggest mistake you've made with regard to design or programming? What did you learn from it?

Chuck: Some 20 years ago I wanted to develop a tool to design VLSI chips. I didn't have a Forth for my new PC, so I thought I'd try a different approach: machine language. Not assembler language, but actually typing the hex instructions.

I built up the code as I would in Forth, with many simple words that interacted hierarchically. It worked. I used it for 10 years. But it was difficult to maintain and document. Eventually I recoded it in Forth and it became smaller and simpler.

My conclusion was that Forth is more efficient than machine language. Partly because of its interactivity and partly because of its syntax. One nice aspect of Forth code is that numbers can be documented by the expression used to calculate them.

Hardware

How should people see the hardware they develop on: as a resource or as a limit? If you think of hardware as a resource, you might want to optimize the code and exploit every hardware feature; if you see it as a limit, you are probably going to write code with the idea that your code will run better on a new and more powerful version of the hardware, and that's not a problem because hardware evolves rapidly.

Chuck: A very perceptive observation that software necessarily targets its hardware. Software for the PC certainly anticipates faster computers and can afford to be sloppy.

But for embedded systems, the software expects the system to be stable for the life of the project. And not a lot of software is migrated from one project to another. So here the hardware is a constraint, though not a limit. Whereas, for PCs, hardware is resource that will grow.

The move to parallel processing promises to change this. Applications that cannot exploit multiple computers will become limited as single computers stop getting faster. Rewriting legacy software to optimize parallel processing is impractical. And hoping that smart compilers will save the day is just wishful thinking.

What is the root of the concurrency problem?

Chuck: The root of the concurrency problem is speed. A computer must do many things in an application. These can be done on a single processor with multitasking. Or they can be done simultaneously with multiple processors.

The latter is much faster and contemporary software needs that speed.

Is the solution in hardware, software, or some combination?

Chuck: It's not hard to glue multiple processors together. So the hardware exists. If software is programmed to take advantage of this the problem is solved. However, if the software can be reprogrammed, it can be made so efficient that multiprocessors are not needed. The problem is to use multiprocessors without changing legacy software. This is the intelligent compiler approach that has never been achieved.

I'm amazed that software written in the 1970s hasn't/can't be rewritten. One reason might be that in those days software was exciting; things being done for the first time; programmers working 18-hour days for the joy of it. Now programming is a 9–5 job as part of a team working to a schedule; not much fun.

So they add another layer of software to avoid rewriting the old software. At least that's more fun than recoding a stupid word processor.

We have access to a big computational power in common computers, but how much actual computing (that is, calculating) are these systems doing? And how much are they just moving and formatting data?

Chuck: You are right. Most computer activity is moving data, not calculating. Not just moving data, but compressing, encrypting, scrambling. At high data rates, this must be done with circuitry so one wonders why a computer is needed at all.

Can we learn something from this? Should we build hardware in a different way?

Don Knuth launched a challenge: check what happens inside a computer during one second of time. He said that what we would discover could change a lot of things.

Chuck: My computer chips recognize this by having a simple, slow multiply. It isn't used very often. Passing data between cores and accessing memory are the important features.

On one hand you have a language that really enables people to develop their own vocabularies and not necessarily think about the hardware presentation. On the other hand, you have a very small kernel that's very much tied to that hardware. It's interesting how Forth can bridge the gap between the two. On some of these machines, is it true that you have no operating system besides your Forth kernel?

Chuck: No, Forth is really standalone. Everything that needs to exist is in the kernel.

But it abstracts away that hardware for people who write programs in Forth.

Chuck: Right.

The Lisp Machine did something similar, but never really was popular. Forth quietly has done that job.

Chuck: Well, Lisp did not address I/O. In fact, C did not address I/O and because it didn't, it needed an operating system. Forth addressed I/O from the very beginning. I don't believe in the most common denominator. I think that if you go to a new machine, the only reason it's a new machine is because it's different in some way and you want to take advantage of those differences. So, you want to be there at the input-output level so you can do that.

Kernighan and Ritchie might argue for C that they wanted a least common factor to make porting easier. Yet you found it easier to port if you didn't take that approach.

Chuck: I would have standard ways of doing that. I would have a word—I think it was `fetchp` maybe—that would fetch 8 bits from a port. That would be defined differently on different computers, but it would be the same function at the stack.

In one sense then, Forth is equivalent to C plus the standard I/O library.

Chuck: Yeah, but I worked with the Standard FORTRAN Library in the early days, and it was awful. It just had the wrong words. It was extremely expensive and bulky. It was so easy to define half a dozen instructions to perform in I/O operation that you didn't need the overhead of a predefined protocol.

Did you find yourself working around that a lot?

Chuck: In FORTRAN, yeah. When you're dealing with, say, Windows, there's nothing you can do. They won't let you have access to the I/O. I have stayed away from Windows most deliberately, but even without Windows, the Pentium was the most difficult machine to put Forth on.

It had too many instructions. And it had too many hardware features like the lookaside buffers and the different kinds of caching you really couldn't ignore. You had to wade your way through, and the initialization code necessary to get Forth running was the most difficult and the most bulky.

Even if it only had to be executed once, I spent most of my time trying to figure out how to do it correctly. We had Forth running standalone on a Pentium, so it was worth the trouble.

The process extended over 10 years probably, partly chasing the changes in the hardware Intel was making.

You mentioned that Forth really supports asynchronous operation. In what sense do you mean asynchronous operation?

Chuck: Well, there's several senses. Forth has always had a multiprogramming ability, a multithreading ability called Cooperative.

We had a word called `pause`. If you had a task and it came to a place where it didn't have anything to do immediately, it would say `pause`. A round-robin scheduler would assign the computer to the next task in the loop.

If you didn't say `pause`, you could monopolize the computer completely, but that would never be the case, because this was a dedicated computer. It was running a single application and all the tasks were friendly.

I guess that was in the old days when all of the tasks were friendly. That's one kind of asynchronism that these tasks could run, do their own thing without ever having to synchronize. One of the features, again, of Forth is that that word `pause` could be buried in lower-level words. Every time you tried to read or write disk, the word `pause` would be executed for you, because the disk team knew that it was going to have to wait for the operation to complete.

In the new chips, the new multicore chips that I'm developing, we're taking that same philosophy. Each computer is running independently and if you have a task on your computer, and another task on the neighbor, they're both running simultaneously but they're communicating with each other. That's the equivalent of what the tasks would've been doing in a threaded computer.

Forth just factors very nicely into those independent tasks. In fact, in the case of the multicore computer, I can use not exactly the same programs, but I can factor the programs in the same way to make them run in parallel.

When you had the cooperative multithreading, did each thread of execution have its own stack, and you switched between them?

Chuck: When you did a task switch, sometimes all you needed to do, depending on the computer, was save the word on top of the stack and then switch the stack pointer. Sometimes you actually had to copy out the stack and load the new one, but in that case, I would make it a point to have a very shallow stack.

Did you deliberately limit the stack depth?

Chuck: Yes. Initially, the stacks were arbitrarily long. The first chip I designed had a stack that was 256 deep because I thought that was small. One of the chips I designed had a stack 4 deep. I've settled now on about 8 or 10 as a good stack depth, so my minimalism has gotten stricter over time.

I would've expected it to go the other way.

Chuck: Well, in my VLSI design application, I do have a case where I'm recursively following traces across the chip, in which case, I have to set the stack depths to about 4,000. To do that might require a different kind of stack, a software-implemented stack. But, in fact, on the Pentium it can be a hardware stack.

Application Design

You brought up the idea that Forth is an ideal language for many small computers networked together—smart dust, for example. For which kinds of applications do you think these small computers are the most appropriate?

Chuck: Communication certainly, sensing certainly. But I'm just beginning to learn how independent computers can cooperate to achieve a greater task.

The multicore computers we have are brutally small. They have 64 words of memory. Well, to put it differently, they have 128 words of memory: 64 RAM, 64 ROM. Each word can hold up to four instructions. You might end up with 512 instructions in a given computer, period, so the task has to be rather simple. Now how do you take a task like the TCP/IP stack and factor it amongst several of these computers in such a way that you can perform the operation without any computer needing more than 512 instructions? That's a beautiful design problem, and one that I'm just approaching now.

I think that's true of almost all applications. It's much easier to do an application if it's broken up into independent pieces as it is trying to do it in serial on a single processor. I think that's true of video generation. Certainly I think it's true of compressing and uncompressing images. But I'm just learning how to do that. We've got other people here in the company that are also learning and having a good time at it.

Is there any field of endeavor where this is not appropriate?

Chuck: Legacy software, certainly. I'm really worried about legacy software, but as soon as you're willing to rethink a problem, I think it is more natural to think of it this way. I think it corresponds more closely to the way we think the brain works with Minsky's independent agents. An agent to me is a small core. It may be that consciousness arises in the communication between these, not in the operation of any one of them.

Legacy software is an unappreciated but serious problem. It will only get worse—not only in banking but in aerospace and other technical industries. The problem is the millions of lines of code. Those could be recoded, say in thousands of lines of Forth. There's no point in machine translation, which would only make the code bigger. But there's no way that code could be validated. The cost and risk would be horrendous. Legacy code may be the downfall of our civilization.

It sounds like you're betting that in the next 10 to 20 years we'll see more and more software arise from the loose joining of many small parts.

Chuck: Oh, yes. I'm certain that's the case. RF communication is so nice. They talk about micro agents inside your body that are fixing things and sensing things, and these agents can only communicate via RF or maybe acoustic.

They can't do much. They're only a few molecules. So this has got to be how the world goes. It's the way our human society is organized. We have six and half billion independent agents out there cooperating.

Choosing words poorly can lead to poorly designed, poorly maintainable applications. Does building a larger application out of dozens or hundreds of small words lead to jargon? How do you avoid that?

Chuck: Well, you really can't. I find myself picking words badly. If you do that, you can confuse yourself. I know in one application, I had this word—I forget what it was now—but I had defined and then I had modified it, and it ended up meaning the opposite of what it said.

It was like you had a word called `right` that makes things go to the left. That was hideously confusing. I fought it for a while and finally renamed the word because it was just impossible to understand the program with that word throwing so much noise into your cognition. I like to use English words, not abbreviations. I like to spell them out. On the other hand, I like them to be short. You run out of short meaningful English words after a while and you've got to do something else. I hate prefixes—a crude way to try to create namespaces so you can use the same old words over and over. They just look to me like a cop out. It's an easy way to distinguish words, but you should've been smarter.

Very often Forth applications will have distinct vocabularies where you can reuse words. In this context, the word does this; in that context, it does something else. In the case of my VLSI design, all of this idealism failed. I needed at least a thousand words, and they're not English words; they're signal names or something, and I quickly had to revert to definitions and weirdly spelled words and prefixes and all of that stuff. It isn't all that readable. But on the other hand, it's full of words like `nand` and `nor` and `xor` for the various gates that are involved. Where possible, I use the words.

Now, I see other people writing Forth; I don't want to pretend to be the only Forth programmer. Some of them do a very good job of coming up with names for things; others do a very bad job. Some come up with a very readable syntax, and others don't think that that's important. Some come up with very short definitions of words, and some have words that are a page long. There are no rules; there's only stylistic conventions.

Also, the key difference between Forth and C and Prolog and ALGOL and FORTRAN, the conventional languages tried to anticipate all possible structures and syntax and build it into the language in the first place. That has led to some very clumsy languages. I think C is a clumsy language with its brackets and braces and colons and semicolons and all of that. Forth eliminated all of that.

I didn't have to solve the general problem. I just had to provide a tool that someone else could use to solve whatever problem they encountered. The ability to do anything and not the ability to do everything.

Should microprocessors include source code so that they can be fixed even decades later?

Chuck: You're right, including the source with microcomputers will document them nicely. Forth is compact, which facilitates that. But the next step is to include the compiler and editor so that the microcomputer code can be examined and changed without involving another computer/operating system that may have been lost. colorForth is my attempt to do that. A few K of source and/or object code is all that's required. That can easily be stored on flash memory and be usable in the far future.

What is the link between the design of a language and the design of a software written with that language?

Chuck: A language determines its use. This is true of human-human languages. Witness the difference between Romance (French, Italian), Western (English, German, Russian) and Eastern (Arabic, Chinese) languages. They affect their cultures and their worldview. They affect what is said and how it's said. Of these, English is particularly terse and increasingly popular.

So too with human-computer languages. The first languages (COBOL, FORTRAN) were too verbose. Later languages (Algol, C) had excessive syntax. These languages necessarily led to large, clumsy descriptions of algorithms. They could express anything, but do it badly.

Forth addresses these issues. It is relatively syntax-free. It encourages compact, efficient descriptions. It minimizes the need for comments, which tend to be inaccurate and distract attention from the code itself.

Forth also has a simple, efficient subroutine call. In C, a subroutine call requires expensive setup and recovery. This discourages its use. And encourages elaborate parameter sets that amortize the cost of the call, but lead to large, complex subroutines.

Efficiency allows Forth applications to be very highly factored, into many, small subroutines. And they typically are. My personal style is one-line definitions—hundreds of small subroutines. In such a case, the names assigned this code become important, both as a mnemonic device and as a way to achieve readability. Readable code requires less documentation.

The lack of syntax allows Forth a corresponding lack of discipline. This, to me, allows individual creativity and some very pleasant code. Others view it as a disadvantage, fearing management loss of control and lack of standardization. I think that's more of a management failure than the fault of the language.

You said "Most errors are in syntax." How do you avoid the other types of errors in Forth programs, such as logic errors, maintainability errors, and bad style decisions?

Chuck: Well, the major error in Forth has to do with stack management. Typically, you leave something on the stack inadvertently and it'll trip you up later. We have a stack comment associated with words, which is very important. It tells you what is on the stack upon entry and what is on the stack upon exit. But that's only a comment. You can't trust it.

Some people did actually execute those and use them to do verification and stack behavior.

Basically, the solution is in the factoring. If you have a word whose definition is one line long, you can read through it thinking how the stack acts and conclude at the end that it's correct. You can test it and see if it works the way you thought it did, but even so, you're going to get caught up in stack errors. The words `dup` and `drop` are ubiquitous and have to be used correctly. The ability to execute words out of context just by putting their input parameters and looking at their output parameters is hugely important. Again, when you're working bottom-up, you know that all of the words you've already defined work correctly because you tested them.

Also, there are only a few conditionals in Forth. There's an `if-else-then` construction, a `begin-while` construct. My philosophy, which I regularly try to teach, is that you minimize the number of conditionals in your program. Rather than having a word that tests something and either does this or that, you have two words: one that does this and one that does that, and you use the right one.

Now it doesn't work in C because the calling sequences are so expensive that they tend to have parameters that let the same routine do different things based upon the way it's called. That's what leads to all of the bugs and complications in legacy software.

In trying to work around deficiencies of the implementation?

Chuck: Yeah. Loops are unavoidable. Loops can be very, very nice. But a Forth loop, at least a colorForth loop, is a very simple one with a single entry and a single exit.

What advice would you give a novice to make programming more pleasant and effective?

Chuck: Well, surely not to your surprise, I would say you should learn to write Forth code. Even if you aren't going to be writing Forth code professionally, exposure to it will teach you some of these lessons and give you a better perspective on whatever language you use. If I were writing a C program, I have written almost none, but I would write it in the style of Forth with a lot of simple subroutines. Even if there were a cost involved there, I think it would be worth it in maintainability.

The other thing is keep it simple. The inevitable trend in designing an aircraft or in writing an application, even a word processor, is to add features and add features and add features until the cost becomes unsupportable. It would be better to have half a dozen word processors that would focus on different markets. Using Word to compose an email is silly;

99% of all of the facilities available are unnecessary. You ought to have an email editor. There used to be such, but the trend seems to be away from that. It's not clear to me why.

Keep it simple. If you're encountering an application, if you're on part of a design team, try to persuade other people to keep it simple. Don't anticipate. Don't solve a problem that you think might occur in the future. Solve the problem you've got. Anticipating is very inefficient. You can anticipate 10 things happening, of which only one will, so you've wasted a lot of effort.

How do you recognize simplicity?

Chuck: There's I think a budding science of complexity, and one of their tenets is how to measure complexity. The description that I like, and I don't know if there's any other one, is that the shortest description or if you have two concepts, the one with the shorter description is the simpler. If you can come up with a shorter definition of something, you come up with a simpler definition.

But that fails in a subtle way that any kind of description depends on the context. If you can write a very short subroutine in C, you might say this is very simple, but you're relying upon the existence of the C compiler and the operating system and the computer that's going to execute it all. So really, you don't have a simple thing; you have a pretty complex thing when you consider the wider context.

I think it's like beauty. You can't define it, but you can recognize it when you see it—simple is small.

How does teamwork affect programming?

Chuck: Teamwork—much overrated. The first job of a team is to partition the problem into relatively independent parts. Assign each part to an individual. The team leader is responsible for seeing that the parts come together.

Sometimes two people can work together. Talking about a problem can clarify it. But too much communication becomes an end in itself. Group thinking does not facilitate creativity. And when several people work together, inevitably one does the work.

Is this valid for every type of project? If you have to write something as feature-rich as OpenOffice.org… it sounds pretty complex, no?

Chuck: Something like OpenOffice.org would be factored into subprojects, each programmed by an individual with enough communication to assure compatibility.

How do you recognize a good programmer?

Chuck: A good programmer writes good code quickly. Good code is correct, compact, and readable. "Quickly" means hours to days.

A bad programmer will want to talk about the problem, will waste time planning instead of writing, and will make a career out of writing and debugging the code.

What is your opinion of compilers? Do you think they mask the real skills of programmers?

Chuck: Compilers are probably the worst code ever written. They are written by someone who has never written a compiler before and will never do so again.

The more elaborate the language, the more complex, bug-ridden, and unusable is the compiler. But a simple compiler for a simple language is an essential tool—if only for documentation.

More important than the compiler is the editor. The wide variety of editors allows each programmer to select his own, to the great detriment of collaborative efforts. This fosters the cottage industry of translating from one to another.

Another failing of compiler writers is the compulsion to use every special character on the keyboard. Thus keyboards can never become smaller and simpler. And source code becomes impenetrable.

But the skills of a programmer are independent of these tools. He can quickly master their foibles and produce good code.

How should software be documented?

Chuck: I value comments much less than others do. Several reasons:

- If comments are terse, they are often cryptic. Then you have to guess what they mean.
- If comments are verbose, they overwhelm the code they're embedded in and trying to explain. It's hard to find and relate code to comment.
- Comments are often badly written. Programmers aren't known for their literary skills, especially if English is not their native language. Jargon and grammatical errors often make them unreadable.
- Most importantly, comments are often inaccurate. Code may change without comments being updated. Although code may be critically reviewed, comments rarely are. An inaccurate comment causes more trouble than no comment. The reader must judge whether the comment or the code is correct.

Comments are often misguided. They should explain the purpose of the code, not the code itself. To paraphrase the code is unhelpful. And if it is inaccurate, downright misleading. Comments should explain why the code is present, what it is intended to accomplish, and any tricks employed in accomplishing it.

colorForth factors comments into a shadow block. This removes them from the code itself, making that code more readable. Yet they are instantly available for reading or updating. It also limits the size of comments to the size of the code.

Comments do not substitute for proper documentation. A document must be written that explains in prose the code module of interest. It should expand greatly the comments and concentrate on literate and complete explanation.

Of course, this is rarely done, is often unaffordable, and is easily lost since it is separate from the code.

Quoting from http://www.colorforth.com/HOPL.html*:*

"The issue of patenting Forth was discussed at length. But since software patents were controversial and might involve the Supreme Court, NRAO declined to pursue the matter. Whereupon, rights reverted to me. I don't think ideas should be patentable. Hindsight agrees that Forth's only chance lay in the public domain. Where it has flourished."

Software patents are still controversial today. Is your opinion about patents still the same?

Chuck: I've never been in favor of software patents. It's too much like patenting an idea. And patenting a language/protocol is especially disturbing. A language will only be successful if it's used. Anything that discourages use is foolish.

Do you think that patenting a technology prevents or limits its diffusion?

Chuck: It is difficult to market software, which is easy to copy. Companies go to great lengths to protect their product, sometimes making it unusable in the process. My answer to that problem is to sell hardware and give away the software. Hardware is difficult to copy and becomes more valuable as software is developed for it.

Patents are one way of addressing these issues. They have proven a wonderful boon to innovation. But there's a delicate balance required to discourage frivolous patents and maintain consistency with prior art/patents. And there are huge costs associated with granting and enforcing them. Recent proposals to reform patent law threaten to freeze out the individual inventor in favor of large companies. Which would be tragic.

CHAPTER FIVE

BASIC

In 1963, Thomas Kurtz and John Kemeny invented BASIC, a general-purpose language intended to teach beginners to program as well as to allow experienced users to write useful programs. Their original goals included abstracting away details of the hardware. The language spread widely after the introduction of microcomputers in the 70s; many personal computers included custom variants. Though the language has moved beyond line numbers and GOTO statements through Microsoft's Visual Basic and Kurtz's BASIC, multiple generations of programmers learned the joy of programming from a language that encouraged experimentation and rewarded curiosity.

The Goals Behind BASIC

What is the best way to learn to program?

Tom Kurtz: Beginning programmers should not have to wade through manuals. Most manuals are far too wordy to retain the attention of new students. Simple coding assignments and easy access to easy-to-use implementations are required, and many examples.

Some educators prefer to teach a language in which programmers need to develop a lot of experience before applying it. You have chosen instead to create a language that any level of programmer can use quickly, where they can improve their knowledge by experience.

Tom: Yes. Once you have learned to program, new computer languages are easy to learn. The first is the hardest. Unless a language is particularly obtuse, the new language will be but a short step from the languages already known. An analogy with spoken languages (which are much more difficult to learn): once you learn your first Romance language, the second is much simpler. First of all, the grammar is similar, there are many words the same, and the syntax is fairly simple (i.e., whether the verb is in the middle, as in English, or at the end).

The simpler the first language, the more easily the average student will learn it.

Did this evolutionary approach guide your decision to create BASIC?

Tom: When we were deciding to develop BASIC (John Kemeny and I back in 1962 or so), I considered attempting to develop simplified subsets of either FORTRAN or Algol. It didn't work. Most programming languages contain obscure grammatical rules that act as a barrier for the beginning student. We tried to remove all such from BASIC.

Several of the considerations that went into the design of BASIC were:

- One line, one statement.

  We couldn't use a period to end a statement, as JOSS did (I believe.) And the Algol convention of a semicolon made no sense to us, as did the FORTRAN Continuation (C).

- Line numbers are GOTO targets.

  We had to have line numbers since this was long before the days of WYSIWYG. Inventing a new concept of "statement label" didn't seem like a good idea to us. (Later, when creating and editing programs became easier, we allowed the user to *not* use line numbers, as long as he didn't use GOTO statements; by that time, BASIC was fully structured.)

- All arithmetic is floating point.

  One of the most difficult concepts for a beginner to learn is why the distinction between type integer and type floating. Almost all the programming languages at the time bowed to the architecture of the most computer hardware, which included floating point for engineering calculations and integer for efficiency.

In handling all arithmetic in floating point, we protected the user from numeric typing. We did have to do some complicated stuff internally when an integer value was required (as in an array subscript) and the user provided a noninteger (as in 3.1). We simply rounded in such cases.

We had similar problems with the difference between binary and decimal fractions. As in the statement:

```
FOR I = 1 TO 2 STEP 0.1
```

The decimal fraction 0.1 is an infinite repeating binary fraction. We had to use a fuzz factor to determine the completion of the loop.

(Some of these binary-decimal considerations were not included in the original BASIC, but were handled in the much more recent True BASIC.)

- A number is a number (is a number).

  No form requirements when entering a number in the code or in data statements. And the PRINT statement produced answers in a default format. The FORMAT statement, or its equivalent in other languages, is quite difficult to learn. And the beginning user might wonder why would he have to learn it—he just wanted to get a simple answer!

- Reasonable defaults.

If there are any complications for the "more advanced" user, they should not be visible to the beginner. Admittedly, there were not many "advanced" features in the original BASIC, but that idea was, and is, important.

The correctness of our approach was borne out by that fact that it took about an hour to teach freshmen how to write simple programs in BASIC. Our training started out with four one-hour lectures, then was reduced to three, then two, and finally to a couple of videotapes.

I once determined that an introductory computer science course could be taught using a version of BASIC (not the original one, but one that included structured programming constructs). The only thing you could not do was to introduce the student to the ideas of pointers and allocated storage!

Another point: in the early days running a program required several steps: Compiling. Linking and loading. Execution. We decided in BASIC that *all* runs would combine these steps so that the user wouldn't even be aware of them.

At that time in the history of computing, most languages required a multiple-pass compiler, which might consume too much valuable computer time. Thus, we compiled once, and executed many times. But small student programs were compiled and executed once only. It did require us to develop a single-pass compiler, and go directly to execution if the compilation stage was without errors.

Also, in reporting errors to the student, we stopped after five errors. I can recall FORTRAN error printouts many pages in length detailing *all* the syntax errors in a program, usually from omitting but one key punctuation at the beginning.

I've seen a BASIC manual from 1964. The subtitle is "the elementary algebraic language designed for the Dartmouth Time Sharing system." What's an algebraic language?

Tom: Well, we were both mathematicians, so naturally there are certain things in the language that look mathematical, for example raising numbers to power and things of this sort, and then the functions that we added were mathematical, like sine and cosine, because we were thinking of students doing calculus using BASIC programs. So there was obviously a bias for numerical calculations in contrast to other languages that were developed at the time such as COBOL, which had a different focus.

What we did was look at FORTRAN at the time. Access to FORTRAN on any of the big IBM computers was through the medium of 80-column punch cards. We were introducing a computer in our campus through the use of teletype machines, which were used as input to computers because they were compatible with phone lines, and we wanted the phone lines to connect the terminals in various places on the campus to the central computer. So all that was done using machinery designed originally for communication purposes such as teletype communication, store and forward messages, and so on. So we did away with punch cards.

Second thing we wanted to do was to get away from the requirements that punch cards imposed on users, which was that things had to be in certain columns on the card, and so we wanted to be something more or less free form that somebody could type on a teletype keyboard, which is just a standard "qwerty" keyboard, by the way, but only with uppercase letters.

That's how the form of the language appeared, something that was easy to type, in fact originally it was space-independent. If you put spaces or you didn't put spaces in what you were typing it didn't make any difference, because the language was designed originally so that whatever you typed was always interpreted by the computer correctly, even if there were spaces or no spaces. The reason for that was that some people, especially faculty members, couldn't type very well.

Space insensitivity made its way into some of the early personal computer versions of BASIC, and that led to some quite funny anomalies about the interpretation of what the person typed.

At Dartmouth there was no ambiguity at all. Only in much later years were spaces required as the language evolved; the ending of a variable name had to be either a space or symbol.

One critic of BASIC said that it is a language designed to teach; as soon as you start writing big programs, they become chaotic. What do you think?

Tom: This is a statement by somebody who hasn't followed the development of BASIC over the years. It's not a baby language. With True BASIC I personally wrote 10,000- and 20,000-line programs, and it expands quite well, and I could write 30,000- or 40,000-line programs and there wouldn't be any problem, and it wouldn't cause the runtime to become inefficient, either.

The implementation of the language is separate from the design of the language.

The design of the language is what the user has to type to get his work done. Once you allow the possibility of libraries, then you can do anything you want. Then it's a question of the implementation of the language whether it supports programs of infinitely large size, and True BASIC does.

This is different from other versions of BASIC. For example Microsoft BASIC and Visual Basic, that is based on it, have some limitations. Other versions of BASIC that have been floating around had other limitations, but those are in the implementation, not in the design of the language.

Which features of True BASIC made it possible for you to write large programs?

Tom: There's only one, the encapsulation, the module. We call our encapsulating structures *modules*.

That was actually standardized by the BASIC Committee, believe it or not, before they went out of business. That happened in the early days of True BASIC. That feature was added to the language standard, and that was about 1990 or so, 1991.

Modern computers have lots of memory and very fast chips and so there's no problem implementing that kind of stuff.

Even though you're back to two passes now in the compiler.

Tom: The linker is also written in True BASIC. It's actually a crude version of True BASIC, or a simplified version of True BASIC. That's compiled into this B code, like the Pascal P code.

To actually do the linking, you execute those B code instructions and there's a very fast interpreter that executes B code instructions. True BASIC, like the original BASIC, is compiled. The original BASIC was compiled into direct machine instructions, in one stage. In True BASIC we compile into B code, and the B code is very simple, so the execution of B code by a very fast C written loop, as it is now, was originally written for the DOS platforms.

That's very fast. It's not as fast as a language designed for speed, but it's pretty darn fast. As I said, there are two-address instructions in the B code, and so it's very fast.

In the early days, interpretation didn't slow things down because we had to do floating point in software. We insisted that True BASIC and original Dartmouth BASIC always dealt with double-precision numbers, so that 99% of the users didn't have to worry about the precision. Now, of course, we use the IEEE standard that's provided automatically by all chips.

Do you think that the only difference between a language designed to teach and one designed to build professional software is that the first is easier to learn?

Tom: No, it's just the way that things developed. C came at an appropriate time and gave access to the hardware. Now the current object-oriented languages that are around, what they are teaching and what the professionals are doing, are derivatives of that environment, and so those languages are very hard to learn.

It means that people who use these derivative languages and are professionally trained and are members of programming teams can put together much more sophisticated applications, such those used to do movies, sounds, and things of this sort. It is just much easier to do that with an object-oriented language like Objective-C, but if that's not your goal, and you just want to write a large application program, you could use True BASIC, which comes from Dartmouth BASIC.

What is the final goal of making a programming language easier to use? Will we ever be able to build a language so simple that every computer user could write his own programs?

Tom: No, a lot of the stuff we based on BASIC at Dartmouth can now be handled by other applications such as spreadsheets. You can do quite complicated calculations with spreadsheets. Furthermore, some of the mathematical applications we had in mind can now be done using libraries of programs put out by professional societies.

The details of the programming language don't really matter because you can learn new languages in one day. It is easy to learn a new language if there is proper documentation. I just don't see what is the need of any new language alleged to be the perfect language. Without a specific field in mind, you can't have a good language; it's a self-contradicting idea! It's like asking what is the best spoken and written language around the world? Is it Italian? English? Or what is it? Could you define one? No, because all written and spoken languages derive from how life is in that place where the language is used, so there is no such thing as the perfect language. There is no perfect programming language, either.

Did you always intend that people would write a hundred very small programs and then call themselves programmers?

Tom: That was our purpose, but the odd thing about it is, as the language grew, without getting too complex, it became possible to write 10,000-line programs. That's because we kept things very simple. The whole idea, and you see, the trick in time sharing is that the turnaround time is so quick, you don't worry about optimizing the program. You worry about optimizing person time.

I had an experience when I was writing a program for the MIT computer several years before we invented BASIC. That was using a symbolic assembly program, SAP, for the IBM 704. I tried to write this program and I tried to do everything that made sense, and I used sense lights to optimize it, so I didn't repeat calculations that weren't necessary. I did everything. Well, the damn thing didn't work and it took me a month to find out that it didn't work, because I went down every two weeks. The turnaround time was two weeks.

I used I don't know how many minutes or hours of computer time in the process. Then the next year when FORTRAN came out, I switched and wrote a FORTRAN program and I think I used five minutes of computer time, all told.

The whole business of optimizing and coding is absolutely wrong. You don't do that. You optimize only if you have to and you do it later. Higher-level languages optimize computer time automatically because you make fewer errors.

That's a point I hear infrequently.

Tom: Computer scientists are kind of stupid in that respect. When we're computer programmers we're concentrating on the intricate little fascinating details of programming and we don't take a broad engineering point of view about trying to optimize the total system. You try to optimize the bits and bytes.

At any rate, that's just an editorial comment. I'm not sure I could back it up.

Did the evolution of the hardware influence the evolution of the language?

Tom: No, because we thought the language was a protection from knowing about the hardware. When we designed BASIC we made it hardware-independent; there is nothing in the language or in the features that came in later that reflects the hardware.

This is not true with some of the early personal computer versions of BASIC, which were based only in a loose sense on what we did at Dartmouth. For example, in one personal computer version of BASIC they had a way to set or interrogate the content of a certain memory location. In our own BASIC at Dartmouth, we never had that. So of course those personal computer BASICs were terribly dependent on the hardware capabilities, and the design of those personal computer languages reflected the hardware that was available to them.

If you were talking to people who did Microsoft BASIC, they would say yes, the features of the language were influenced by the hardware, but this didn't happen at Dartmouth with the original BASIC.

You chose to perform all arithmetic as floating point to make things easier for the user. What is your opinion on the way modern programming languages handle numbers? Should we move to an exact form of representation using arbitrary-precision numbers, where you consider them as a sort of "array of digits"?

Tom: There are lots of ways to represent numbers. It is true that most languages at that time, and modern languages as well, reflect the availability of the type of number representations that are available on today's hardware.

For example, if you program in C today, there are number types that correspond to the numeric representation available on hardware, such as single-precision floating point, double-precision floating point, single-precision integer, double-precision integer, etc. Those are all aspects of the C language because it was designed to get at the hardware, so they have to provide access to whatever the number representations are in the computer.

Now, what numbers can be represented in computers? Well, in a fixed-length number of binary digits, binary bits—with which most computers work—are at least a finite number of decimal digits, you have a limitation on the type and numbers you can represent, and that's well known to lead to certain types of rounding errors.

Some languages provide access to an unlimited precision, like 300 decimal digits, for example, but they do that with software by representing very large numbers as potentially infinite arrays of digits, but that's all done by a software and consequently is very slow.

Our approach in BASIC was simply to say a number is a number, "3" is a number but also "1.5" is a number. We haven't bothered our students with that distinction; whatever they put as a number, we tried our best to represent that number in the floating-point hardware that was available on the machine.

One thing to say about that is when we were first considering which computers to get (of course we ended up with the GE computer in 1964), we insisted that the computer had floating-point hardware because we didn't want to mess around with having to do software arithmetic, and so that's how we represented the numbers. Of course there is a certain imprecision in that, but that's what you have to live with.

Were the GOTO and the GOSUB statements just a choice given the hardware at the time? Should modern programming languages provide them as well?

Tom: I don't think the hardware was the issue; it's irrelevant.

Some structured languages required it, but that was in the old days, 20 or 30 years ago, so I don't really think that's an issue.

The thing was important at the time because that was how people wrote programs for computers in machine language and assembly language. When we did BASIC the idea of structured programming had not yet surfaced; also, we patterned BASIC after FORTRAN, and FORTRAN had the GOTO statement.

During the evolution of BASIC, what criteria did you use when considering new features to add to the language?

Tom: Well, whatever was needed at the time—nothing very theoretical.

For example, one of the things we did after BASIC saw the light of day in early 1964 was to add the ability to handle nonnumerical information, strings of character information. We allowed character strings so that when people were writing programs such as to play games, they could type "yes" or "no" instead of "1" or "0". In the original BASIC, "1" meant "yes," and "0" meant "no," but very soon we added the ability to handle strings of characters. And that was just because it was needed.

Compiler Design

When you wrote the first version of BASIC, you were able to write a single-pass compiler while everyone else was doing a multipass compiler. How did you do that?

Tom: It's very simple, if the design of the language is relatively simple. A lot of languages are simple in that respect. Everything was known, and the only thing we had to put off to what we call the pass and a half was filling in for forward transfers. That was the only thing that really prevented a complete single-pass compiler.

In the first hundred lines of a program you have a GOTO to something in the first thousand lines. It's a linking stage then.

Tom: That's what we did. It was the equivalent of the linking list. Now, we didn't actually use a linked list structure in the assembly language of the computer we were working with, but it was basically that. It might have been a little table that was set up with addresses that are filled in later.

Were you able to parse and generate code at the same point then?

Tom: Yes. The other part about it was that the language was deliberately made simple for the first go-round so that a single-pass parsing was possible. In other words, variable names are very limited. A letter or a letter followed by a digit, and array names, one- and two-dimensional arrays were always single letters followed by a left parenthesis. The parsing was trivial. There was no table lookup and furthermore, what we did was to adopt a simple strategy that a single letter, or a single letter followed by a digit, gives you what, 26 times 11 variable names. We preallocated space, fixed space for the locations for the values of those variables, if and when they had values.

We didn't even use a symbol table.

Did you require variable declarations?

Tom: No, absolutely not. In fact, arrays always were single letters followed by left parenthesis, so that was in fact the declaration. Let me see if I can remember this correctly. If you used an array, like you used `a(3)`, then it was automatically an array from, oh, let's see, I think it was 0 to 10. Automatic default declarations, in other words, and starting at 0 because, being mathematicians, when you represent the coefficients of a polynomial, the first one has a 0 subscript.

Did you find that simple to implement?

Tom: Trivial to implement. In fact, there are a lot of things in compiler writing that are not too hard at all. Even later on when a more advanced version of BASIC was floating around that used a symbol table lookup, but that's not so hard, either.

It's the optimizations that hurt.

Tom: We didn't worry about optimization, because 99% of all the programs that were being written by students and by faculty members at that time were little teeny, little trivial things. It didn't make any sense to optimize.

You've said that polymorphism implies runtime interpretation.

Tom: I believe that's true, but nobody has challenged me on that statement because I haven't discussed it. Polymorphism means that you write a certain program and it behaves differently depending on the data that it operates on. Now, if you don't pull that in as a source program, then at execution time that piece of program doesn't know what it's doing until it actually starts executing, that's runtime interpretation. Am I wrong on that?

Consider Smalltalk, where arguably you have the source available. If you make really late binding decisions, does that count as runtime binding?

Tom: That's a tricky question. There's early binding, late binding, and runtime binding. It's really tricky, and I imagine you can figure out ways of getting around this.

For example, suppose you're writing a sorting routine. If you're sorting numbers, the comparison between which number is less and so on is obvious. If you're sorting character strings, then it's less obvious, because you don't know whether you want ASCII sorting or whether you want dictionary ordering or whether you want some other ordering.

If you're writing a sorting routine, you know which one you want, so that's how you make your comparison. If you're writing a general purpose sorting routine, then you have to call a subroutine or do something like that to determine whether item A is less than item B, whatever that is. If you're trying to sort keys to records or something, then you have to know the ordering of whatever it is you're sorting. They may be different kinds of things. Sometimes they may be character strings, but think of what the possibilities are. When you write the sorting algorithm, you don't know any of that stuff, which means it has to be put in later. If it's done at runtime, of course, then it's runtime interpretation.

That can be done efficiently, don't get me wrong, because you can have a little program, a subroutine, and then all the person has to do is, in the subroutine, to write the rules for ordering the elements that he's sorting. But it isn't automatic.

You don't get polymorphism for free. You have to write the polymorphic variants.

Tom: Somebody has to worry about it.

The other thing that the object-oriented people talk about is inheritance. That's only important if you have data typing in your language. I've read the introductions to a number of object-oriented books, and they talk about a guy writing a circle routine. Somebody else might use it for some other purpose, but that's extremely rare. The problem that I've always felt about stuff like that is that if you want to write a routine that's general-purpose enough that other people might want to use it, then you've got to document the hell out of it, and you have to make it available. I mean there's a whole raft of considerations. You're almost writing a complete application with documentation.

For the kind of programs I do, that's overrated. I don't know what happens in the industry. That's another matter.

Would you call that idea of cheap and easy code reuse premature generalization?

Tom: It's an idea that may have relevance in the programming profession, but it does not have relevance to the wider group of amateurs who might write programs. As a matter of fact, most people don't write programs these days. Much of what we used to write programs for is now done by an application that you can buy or you can put it into a spreadsheet, or whatever. Having nonprofessional programmers, people in other fields, write programs is not done very much anymore.

One of the things that bothers me about the education of programming primarily in secondary schools, where they have an advanced placement in computer science, is that it's much too complicated. I don't know what languages they teach these days, I haven't looked at it.

I once looked at how I would structure a first college course in computer science using BASIC. It could do practically everything I'd ever want to do in a beginning computer science course except deal with pointers and allocated storage. That's sort of a complexity. If you use Pascal for the language, you may have to get into pointers and allocated storage when people don't even know what a computer program is, but that's neither here nor there. I never pushed my views. I'm one against many.

People are starting to believe that you don't have to deal with allocated memory and pointers much anymore unless you're writing virtual machines. Those who write compilers do, but that's our job.

Tom: Let the compiler do it; you don't have to do it.

We got portability in True BASIC. A couple of young men who were really brilliant did the design. I just was with the company and did application programming. They designed an intermediate language that was in the fashion of the P code of Pascal. Instead of being two address, it was three address, because it turned out that practically all instructions in BASIC are three address, LET A = 3. That's three things, the opcode and the two addresses. Then they built a compiler using BASIC itself, and built a very crude support to actually compile that compiler. The compiler itself is written in True BASIC, and it runs on any machine for which there is a True BASIC engine, which we call the interpreter.

The language is interpreted at the execution level, not at the scanning level. So there's three stages in the execution of the program. The first is the compiler stage, the second is the linking/loading stage, and the third is the execution. But the user doesn't know that. The user just types run or hits run or something and bang, it happens.

The compiled code is also machine-independent. It can transport that across boundaries.

It's really quite a sophisticated language environment. We were on multiple platforms, four or five different platforms for a while, but most platforms lived a short time and died, of course. Now there's only two major platforms or three major platforms left: Unix, Microsoft, and, for us, Apple—an interesting platform because Dartmouth was always an Apple school.

Doing the porting to those platforms turned out to be a dog. The windowing support and the gadgets and the buttons and all that kind of stuff, they all do it differently, and you have to get down to the detail of how they do it. Sometimes they do it at a very low level, so you have to build all that stuff up yourself.

The old, original Mac, it had a Mac toolbox. For a while, we used a layering software, XVT out of Boulder, Colorado, which claimed to target Windows and also the classic Mac. We were able to get some mileage out of that. Before the company went defunct, the programmer put out a version for Windows; it goes directly to the Windows application environment.

The trouble with those is that when we have a single programmer doing all of that stuff, it takes a while, and new versions of the operating system come out and you run across new bugs and have to track them down. It was almost impossible for a small outfit like we were. At one point we had three programmers, then down to two, and then down to one. That's really just too much for one programmer to handle.

The underlying code, now that it's largely C, contains tons of `#ifdefs` in it.

Language and Programming Practice

What is the link between the design of a language and the design of a software written with that language?

Tom: Very tight. Most languages were designed with specific types of software in mind. A prime example was APT, a language for controlling Automatic Programmed Tools.

You added the REM statement for comments in the early days. Has your opinion on comments and software documentation changed over the years?

Tom: No, it's a kind of self-defense mechanism. When I write programs in True BASIC, I do add comments to remind me whatever I was thinking when I wrote the code. So I think that comments play a role, and the role is different depending on what kind of programs you are writing, whether you work in a group or no other people read your code. I believe in comments but only insofar as they are necessary.

Do you have any suggestions for people writing software in teams?

Tom: No, because we have never done it. All the software we have done in our environment has been solo work. In True BASIC we had maybe two or three people writing code, but they were really working on completely separate projects. I just don't have any experience working in teams.

You had a time-sharing machine, so you suggested that users should plan their session at the teletype before sitting there. The motto was: typing is no substitute for thinking. Is this true today?

Tom: I think probably that thinking does take place. When a major company is going to develop a new software product, they do a lot of thinking before it, so I think that's done.

One of the things I do personally is not thinking too much ahead but just start writing the program. Then I will discover that it is not quite working out, so I will scrap the whole thing and start over. That's the equivalent to thinking. I usually start coding just to see what the problems are going to be, and then throw that version away.

It is important to think about what you are doing—very important. I am not sure, but I think Richard Hamming stated that "typing is no substitute for thinking." Those are the early days of computing and very few people knew how to do it, so there was a lot of advice like that floating around.

What is the best way to learn a new programming language?

Tom: Once one knows how to program, and knows the concepts (i.e., how storage is allocated), learning a new language is straightforward if one has access to a reference manual, and a decent implementation (i.e., compiler). I've done it many times.

Attending a class is pretty much a waste of time.

Any programmer worth her salt will know many languages in her professional lifetime. (I probably have used more than 20 in mine.) The way to learn new languages is to read the manual. With few exceptions, most programming languages are similar in structure and in the way they operate, so new languages are fairly easy to learn, if there is a reasonable manual available.

Once you get over the jargon hurdle (what does *polymorphism* mean?), things are really fairly simple.

One problem with today's programming style is that there are no manuals—just interface building tools. They are designed so that programmers don't have to type, letter by letter, many of the instructions, but behave like the engineers' CAD and CAM tools. To old-time programmers like me, that is anathema—I want to type all the code, letter by letter.

There have been attempts in the past to simplify the typing (for poor typists or students) by providing macros (such as a single keystroke for the keyword LET), but they never caught on.

I am now attempting to learn a language that is supposedly "object-oriented." No reference manual exists, at least that I have found. The manuals that are available develop what appear to be almost trivial examples, and spend perhaps 90% of the space pointing out how OOP is such a superior "religion." I have friends who took a C++ course, and it was a disaster from a pedagogical point of view. My opinion is that OOP is one of the great frauds perpetrated on the community. All languages were originally designed for a certain class of users—FORTRAN for extended numerical computations, etc. OOP was designed so that its clients could claim superior wisdom for being on the "inside." The truth of the matter is that the single most important aspect of OOP is an approach devised decades ago: encapsulation of subroutines and data. All the rest is frosting.

Language Design

Do you think Microsoft's current Visual Basic is a full-fledged object-oriented language, and if so, do you approve of this aspect of it (given your dismissal of the object-oriented paradigm)?

Tom: I don't know. With a few simple experiments, I found Visual Basic relatively easy to use. I doubt that anyone outside of Microsoft would define VB as an OOL. As a matter of fact, True BASIC is just as much object-oriented as VB, perhaps more so. True BASIC included modules, which are collections of subroutines and data; they provide the single most important feature of OOP, namely, data encapsulation. (True BASIC does not have inherited types, since it doesn't have user-defined types, other than array dimensions. Hardly any language has polymorphism, which, in fact, implies runtime interpretation.)

One of the things that I've asked many of these designers of languages and systems is to what degree they like this notion of a mathematical formalism. Take Scheme, which expresses the lambda calculus very effectively. You have six primitives and everything is just beautifully built on top of that. That seems like the mathematician approach.

Tom: Yeah, that's very interesting. That's an interesting mathematical problem, but if you're designing a computer language, you don't have to do that stuff, because every computer language that I've ever seen is much simpler than that. Even FORTRAN. Algol is simple; it uses recursive definitions, but that's fairly simple and straightforward.

I never studied the theory of programming languages, so I can't make any more comments than that.

Do you consider the people who will use a language and the biggest problems they're going to have to solve?

Tom: Yeah. The biggest problem for the people that we were designing for was remembering the language from week to week, but they only wrote one program every two weeks. We also wanted a programming language and a system environment that we could teach in a matter of a couple of hours, so you don't have to take a course.

That's how I and a lot of my peers learned to program. We had Microsoft BASIC on the PCs of the early 80s—the Commodore 64 and the Apple II. They were line BASICs with subroutines, but not much else.

Tom: There are oddities floating around. They actually had some tricky stuff to it. For example, I used Apple Soft BASIC. I don't know if Microsoft did it or whether somebody else did it, but all of those were copied from Dartmouth BASIC startup. They introduced the idea of a multicharacter variable name, but they didn't parse it correctly. If you happened to have a keyword buried inside your multiple-character variable name, it would throw the thing off.

Was this because of whitespace insensitivity?

Tom: No, because they claimed to have multicharacter variable names, but they didn't. They faked it. If you had a multiple-character variable name which was, let's say, TOT, they would recognize the TO as a keyword. It was a marketing gimmick. Those languages, little features were designed for the market. They thought that multiple-character variable names would be a good gimmick. People that used the language managed to get around that by not using multiple characters very much.

That's not a process that's discoverable reading the manual.

Tom: The errors in it are not, no.

How did whitespace insensitivity come about?

Tom: The only thing I know about it was published, and the reason for space insensitivity is partly because John Kemeny was a poor typist. I don't know if that was really the reason. We codesigned the language, but a feature like that is something that he did.

You made this comment in the context of the discussion about the student who wrote a PL/1 compiler, and the first time he ran it, it worked.

Tom: That's Phil Koch, and he's an Apple fellow. He's retired from Apple now, living in Maine. He was an astonishing programmer. It took him a long time and he read code religiously.

If there is one lesson you'd like people to learn from your vast and varied experiences over the years, what is that?

Tom: Make it easy for your users to use your software.

You can say user friendly if you want, but part of that is that the industry has defined user friendly to be, in my view, condescending. The real issue on user friendliness is to have reasonable defaults in whatever application you're doing, so the person who's just come to it fresh doesn't have to learn about all of the variations and degrees of freedom that are possible. He or she can just sit down and start to do it. Then if they want to do something different, make it relatively easy to get at that.

In order to do that, you have to have some sort of an idea of what your user base is going to be.

I've used Microsoft Word frequently, but by my standards, it's not user friendly at all. Then Microsoft came out with that Bob thing about 10 years ago, and that was the wrong idea. They didn't understand what user friendly really means.

Some applications, I think, are user friendly, but the big thing now is website design. People that design websites, sometimes they do a good job and sometimes they don't do a good job. If you go to a website and you can't figure out what to do to get more information, that's a lousy design.

That stuff's hard to teach.

Ben Shniederman, a specialist in human factors in computer science at the University of Maryland, actually did some studies* that suggested that what we had chosen in BASIC for our structures for DO, LOOP, and IF were easier in the sense of user friendliness than some of the other structures that were using the other languages, like the semicolon in Algol or Pascal to end a sentence.

People normally don't use semicolons to end sentences, so that's something that you have to learn specifically. I remember in FORTRAN, for example, there were places where a comma is needed and places where a comma isn't needed. As a result, there was a bug in a program down at the space station in Florida where they lost a rocket because there was a missing comma. I think Ed Tufte actually documented that. Try to stay away from stuff that's possibly ambiguous.

* Shneiderman, B. "When children learn programming: Antecedents, concepts, and outcomes," *The Computing Teacher*, volume 5: 14–17 (1985).

I keep saying to the world at large, Kemeny and I failed because we didn't make other people's computers user friendly, but we did a good job with our own students because for 20 or so years, our students were going out and getting very cushy jobs in the industry because they knew how to do things.

That is a good type of success to have.

Tom: If you're a teacher, that's really the main thing.