The A-Z of Programming Languages

(interviews with programming language creators)

Computerworld, 2008-2009[1]

[1] When the table of contents is being read using a PDF viewer, the titles link to the Web pages of the original publications, and the page numbers are internal links

Ada: S. Tucker Taft

S. Tucker Taft is a Chairman and CTO of SofCheck. Taft has been heavily involved in the Ada 1995 and 2005 revisions, and still works with the language today as both a designer and user.
Computerworld spoke to Taft to learn more about the development and maintenance of Ada*, and found a man deeply committed to language design and development*

How did you first become involved with Ada?

After graduating in 1975, I worked for Harvard for four years as the 'system mother' for the first Unix system outside of Bell Labs. During that time I spent a lot of time with some of the computer science researchers, and became aware of the DOD-1 language design competition.

I had been fascinated with programming language design for several years at that point, and thought it was quite exciting that there was a competition to design a standard language for mission-critical software. I also had already developed some strong opinions about language design, so I had some complaints about all of the designs.

In September of 1980, a year after I left my job at Harvard, I returned to the Boston area and ended up taking a job at Intermetrics, the company responsible for the design of the Red language, one of the four semifinalists and one of the two finalists for DOD-1. By that time [the language was] renamed to Ada in honor of Lady Ada Lovelace, daughter of Lord Byron and associate of Charles Babbage.

Although Intermetrics had shortly before lost the competition to Honeywell-Bull-Inria, they were still quite involved with the overall process of completing the Ada standard, and were in the process of bidding on one of the two major Ada compiler acquisitions, this one for the Air Force. After a 6-month design period and 12-month public evaluation, the Intermetrics design was chosen over two others and I became first the head of the *Ada Program Support Environment* part, and then ultimately of the Ada compiler itself.

One of the requirements of the Air Force *Ada Integrated Environment* contract was to write the entire compiler and environment in Ada itself, which created some interesting bootstrap problems. In fact, we had to build a separate boot compiler in Pascal, before we could even compile the real compiler. By the time we delivered, we had written almost a million lines of Ada code, and had seen Ada go from a preliminary standard to a Military standard (MIL-STD-1815), to an ANSI standard (Ada 83), and finally to an ISO standard (ISO 8652, Ada 87). I also had to go through the personal progression of learning the language, griping about the language, and then finally accepting the language as it was, so I could use it productively.

However, in 1988 the US Department of Defense announced that they were beginning the process to revise the Ada standard to produce Ada 9X (where X was some digit between 0 and 9). I quickly resurrected all my old gripes and a few new ones, and helped to write a proposal for Intermetrics to become the *Ada 9X Mapping/Revision Team* (the government's nomenclature for the language design team). This time the Intermetrics team won the competition over several other teams, including one that included Jean Ichbiah, the lead designer of the original Ada 83 standard. I was the technical lead of the Intermetrics MRT team, with Christine Anderson of the Air Force as the manager of the overall Ada 9X project on the government side.

What are the main differences between the original Ada and the 95 revision?

The big three Ada 95 language revisions were hierarchical libraries, protected objects, and object-oriented programming. Hierarchical libraries referred to the enhancement of the Ada module namespace to take it from Ada 83's simple flat namespace of library units, where each unit had a single unique identifier, to a hierarchical namespace of units, with visibility control between parent and child library unit.

Protected objects referred to the new passive, data-oriented synchronization construct that we defined to augment the existing active message/rendezvous-oriented task construct of Ada 83.

Object-oriented programming was provided in Ada 95 by enhancing an existing derived-type capability of Ada 83, by supporting type extension as part of deriving from an existing type, as

well as supporting run-time polymorphism with the equivalent of virtual functions and run-time type tags.

What prompted the Ada revision in 95?

ISO standards go through regular revision cycles. Generally every five years a standard must be reviewed, and at least every ten years it must be revised. There were also some specific concerns about the language, though generally the language had turned out to be a reasonably good fit to the needs of mission-critical software development.

In particular, Ada's strong support for abstract data types in the guise of packages and private types had emerged as a significant step up in software engineering, and Ada's run-time checking for array bounds and null pointers had helped catch a large class of typical programming errors earlier in the life-cycle.

Was there a particular problem you were trying to solve?

Object-oriented programming was growing in popularity at the time, though it was still not fully trusted by much of the mission-critical software development community. In addition, the Ada 83 tasking model was considered elegant, but did not provide the level of efficiency or control that many real-time system developers would have preferred.

Once the Ada 9X revision process began, a requirements team was formed to solicit explicit comments from the Ada community about the language, both in terms of things to preserve and things to improve.

Have you faced any hard decisions in your revision of Ada?

Every language-design decision was pretty hard, because there were many goals and requirements, some of which were potentially conflicting. Perhaps the most difficult decisions were political ones, where I realized that to achieve consensus in the language revision process among the ISO delegations, we (the design team) would have to give up some of our personal favourite revision proposals.

Are you still working with the language now and in what context?

Yes, I am still working with the language, both as a user and as a language designer. As you may know the newest version of the language, known as Ada 2005, just recently achieved official standardization. The Ada 2005 design process was quite different from the Ada 95 process, because Ada 2005 had no Department of Defense supported design team, and instead had to rely on strictly voluntary contributions of time and energy.

Nevertheless, I am extremely proud of the accomplishments of the Ada 2005 design working group. We managed to round out many of the capabilities of Ada 95 into a language that overall I believe is even better integrated, is more powerful and flexible, while also being even safer and more secure.

Would you have done anything differently in the development of Ada 95 or Ada 2005 if you had the chance?

The few technical problems in the development of Ada 95 that emerged later during use were either remedied immediately, if minor, through the normal language maintenance activities ('we couldn't have meant that ... we clearly meant to say this'). Or if more major, were largely addressed in the Ada 2005 process. From a process point of view, however, I underestimated the effort required in building international consensus, and in retrospect I should have spent more time establishing the rationale for revision proposals before springing them on the panel of distinguished reviewers and the ISO delegations.

Are you aware of any of the Defence projects for which the language has been used?

Ada was mandated for use by almost all significant Defense department software projects for approximately 10 years, from 1987 to 1997, and there were a large number of such projects. In the early years there were real challenges because of the immaturity of the Ada compilers. In the later years, in part because of the early difficulties, there were a number of projects that applied and received waivers to allow them to use other languages. Nevertheless, in the middle years of 1989 to 1995 or so, there was a boom in the use of Ada, and much of it was quite

successful.

As far as specific projects, the Apache helicopter and the Lockheed C-130J (Hercules II Airlifter) are two well-known examples. The Lockheed C-130J is particularly interesting because it was developed using a formal correctness-by-construction process using the SPARK Ada-based toolset from Praxis High Integrity Systems. The experience with that process was that, compared to industry norms for developing safety-critical avionics software, the C-130J development had a 10 times lower error rate, four times greater productivity, half as expensive a development process, and four times productivity increase in a subsequent project thanks to substantial reuse. NASA has also used Ada extensively for satellite software, and documented significantly higher reuse than their prior non-Ada systems.

In general, in study after study, Ada emerged as the most cost effective way to achieve the desired level of quality, often having an order-of-magnitude lower error rates than comparable non-Ada systems after the same amount of testing.

Can you elaborate more on the development of the Static Interface Analysis Tool (SIAT) for Ada on behalf of the NASA Space Stations IV&V?

The SIAT project was an early attempt to create a browser-based tool for navigating through a complex software system. The particular use in this case was for analyzing the software designed for the large network of computers aboard the International Space Station. It turns out that these systems have a large number of data interfaces, where one computer would monitor one part of the Space Station and report on its state to other computers, by what amounted to a large table of global variables. The SIAT tool was designed to help ensure that the interfaces were consistent, and that data flowed between the computers and these global variable tables in an appropriate way.

Are you aware of why the Green proposal was chosen over the Red, Blue and Yellow proposals at the start of Ada's development?

The Green proposal reached a level of stability and completeness earlier than the other designs, and Jean Ichbiah did an excellent job of presenting its features in a way that the reviewers could understand and appreciate. Although there were flashes of brilliance in the other designs, none of them achieved the polish and maturity of the Green design.

Did you ever work closely with Jean Ichbiah? If so, what was the working relationship like and what did you do together?

I worked on and off with Jean during the final days of the Ada 83 design, and during some of the Ada maintenance activities prior to the start of the Ada 9X design process.

Jean was busy running his own company at the start of the Ada 9X process, but did end up joining the process as a reviewer for a period during 1992 and 1993.

As it turned out, Jean and I had quite different views on how to design the object-oriented features of the updated language, and he ultimately left the project when it was decided to follow the design team's recommended approach.

In your opinion, what lasting legacy have Ada and Ada 95 brought to the Web?

I believe Ada remains the benchmark against which all other languages are compared in the dimension of safety, security, multi-threading, and real-time control. It has also been a source for many of the advanced features in other programming languages. Ada was one of the first widely-used languages to have a language construct representing an abstraction (a package), an abstract data type (a private type), multi-threading (tasks), generic templates, exception handling, strongly-typed separate compilation, subprogram inlining, etc. In some ways Ada was ahead of its time, and as such was perceived as overly complex. Since its inception, however, its complexity has been easily surpassed by other languages, most notably C++, while its combination of safety, efficiency, and real-time control has not been equaled.

Where do you envisage Ada's future lying?

As mentioned above, Ada remains the premier language for safety, security, multi-threading, and real-time control. However, the pool of programmers knowing Ada has shrunk over the years due to its lack of success outside of its high-integrity niche. This means that Ada may

remain in its niche, though that niche seems to be growing over time, as software becomes a bigger and bigger part of safety-critical and high-security systems. In addition, the new growth of multi-core chips plays to Ada's strength in multi-threading and real-time control.

I also think Ada will continue to play a role as a benchmark for other language design efforts, and as new languages emerge to address some of the growing challenges in widely distributed, massively parallel, safety- and security-critical systems, Ada should be both an inspiration and a model for their designers.

Where do you see computer programming languages heading in the future, particularly in the next 5 to 20 years?

As mentioned above, systems are becoming ever more distributed, more parallel, and more critical. I happen to believe that a well-designed programming language can help tame some of this growing complexity, by allowing programmers to structure it, abstract it and secure it.

Unfortunately, I have also seen a large number of new languages appearing on the scene recently, particularly in the form of scripting languages, and many of the designers of these languages seem to have ignored much of the history of programming language design, and hence are doomed to repeat many of the mistakes that have been made.

Do you have any advice for up-and-coming programmers?

Learn several different programming languages, and actually try to use them before developing a religious affection or distaste for them. Try Scheme, try Haskell, try Ada, try Icon, try Ruby, try CAML, try Python, try Prolog. Don't let yourself fall into a rut of using just one language, thinking that it defines what programming means.

Try to rise above the syntax and semantics of a single language to think about algorithms and data structures in the abstract. And while you are at it, read articles or books by some of the language design pioneers, like Hoare, Dijkstra, Wirth, Gries, Dahl, Brinch Hansen, Steele, Milner, and Meyer.

Is there anything else that you'd like to add?

Don't believe anyone who says that we have reached the end of the evolution of programming languages.

ASP: Microsoft

ASP is Microsoft's server-side script engine and Web application framework ASP.NET, used to build dynamic Web sites, applications and Web services

Why was ASP created and what problem/s was it trying to solve?

Active Server Pages (ASP) was initially created to address the challenge of building dynamic Web sites and Web-based business solutions. It was first released with IIS 3.0 (Internet Information Server) in 1996.

Creating and updating static Web sites was a very time consuming task that was prone to human error. In order to avoid mistakes, every page would require careful attention during changes. Furthermore, the potential use of Web sites was very limited using HTML exclusively. There needed to be an efficient way to change content quickly, in real time. ASP enabled the easy integration of databases as well as more advanced business and application logic that the Web is known for today.

Explain the early development of ASP.NET. Who was involved, and what difficult decisions had to be made?

Scott Guthrie is one of the original creators of Microsoft's ASP.NET and, today, is the Corporate Vice President of the Microsoft Developer Division. The early development of ASP.NET focused on developer productivity and enabling powerful, Web-based solutions. The key goal was to help make it easier for traditional developers who had never done Web development before to be successful in embracing this new development paradigm.

ASP.NET was a breakthrough technology that fundamentally changed the way developers approached and delivered Web sites – bringing it more in line with traditional software development.

Building a brand new Web application framework was a difficult decision to make, especially since many customers had already adopted ASP. We felt it was the best approach, since it provided customers with one robust and consistent development platform to build software solutions. A Web developer could now reuse his existing skill set to build desktop or mobile applications.

When we released ASP.NET, we did not want to force customers to upgrade. As a result, we ensured that ASP would work in each subsequent release of IIS. Today, we still continue to support the ASP runtime, which was included as part of IIS7 in Windows Server 2008.

What is the difference between ASP and ASP.NET and why would developers choose one over the other?

ASP and ASP.NET are both server-side technologies and the similarities basically stop there. If a developer is interested in writing less code, we would recommend ASP.NET. There are a myriad of other reasons too, including:

- Great tool support provided by the Visual Studio family and Expression Studio, which makes developers more productive and working with designers much easier.
- ASP.NET AJAX integrated in the framework, which allows a better end-user experience.
- Breadth of the .NET Framework, which provides a wealth of functionality to address both common scenarios and complex ones too.

I would encourage a developer to visit `asp.net` to find out more. A key thing to consider is that ASP.NET is the focus for Microsoft and we are not making any new investments in ASP. I'd highly encourage anyone to use ASP.NET over ASP.

Given a second chance, is there anything Microsoft could have done differently in the development of ASP.NET?

ASP.NET was created to meet the needs of our customers building Web solutions. As with any incubation or v1 product, the biggest change we would have made is to have more transparent and customer integrated product development – much like we have today. Discussion with customers allows us to be better equipped to make decisions that affect them. For example,

ASP.NET MVC (Model-View-Controller) was a request from customers interested in test driven development.

The MVC design pattern is decades old, but the concept can still be applied to the design of today's Web applications. The product team released the first preview at the end of last year, which received a lot of positive feedback. Developers interested in the release wanted more and couldn't wait to try the latest updates. In March, the product team published the source code for ASP.NET MVC on Codeplex and decided to have interim, frequent releases. This allows developers to access the latest bits and provide feedback that influences the first, official release. The community can expect to see similar transparency with other features too.

Why was the decision made to target ASP.NET to IIS and Windows servers? Was this an architectural or business decision? Is it likely that we will ever see a free (possibly open source) official Apache module supporting ASP.NET?

Microsoft is in the business of selling servers so our decision to focus on our products is obvious. We believe that by doing so we can provide the most productive, scalable, secure, and reliable solution for delivering Web applications with deeply integrated features across the Web server, the database, the tools (Visual Studio), and the framework (.NET). ASP.NET is part of the freely available .NET Framework today and we offer free tools like Visual Web Developer Express for anyone to easily get started.

What lasting legacy has ASP brought to the Web?

Never underestimate the value of getting the job done. Even if there is a new Web application framework, we know that some customers are happy with what ASP already provides. We recognize the choice to stay with ASP, and that is why we are continuing our support for the ASP runtime. However, we do believe that continued investments in our new .NET-based server platform will provide developers the best platform choice moving forward.

AWK: Alfred Aho

Computer scientist and compiler expert Alfred V. Aho is a man at the forefront of computer science research. He has been involved in the development of programming languages from his days working as the vice president of the Computing Sciences Research Center at Bell Labs to his current position as Lawrence Gussman Professor in the Computer Science Department at Columbia University.
As well as co-authoring the 'Dragon' book series, Aho was one of the three developers of the AWK pattern matching language in the mid-1970s, along with Brian Kernighan and Peter Weinberger.
Computerworld recently spoke to Professor Aho to learn more about the development of AWK

How did the idea/concept of the AWK language develop and come into practice?

As with a number of languages, it was born from the necessity to meet a need. As a researcher at Bell Labs in the early 1970s, I found myself keeping track of budgets, and keeping track of editorial correspondence. I was also teaching at a nearby university at the time, so I had to keep track of student grades as well.

I wanted to have a simple little language in which I could write one- or two-line programs to do these tasks. Brian Kernighan, a researcher next door to me at the Labs, also wanted to create a similar language. We had daily conversations which culminated in a desire to create a pattern-matching language suitable for simple data-processing tasks.

We were heavily influenced by GREP, a popular string-matching utility on Unix, which had been created in our research center. GREP would search a file of text looking for lines matching a pattern consisting of a limited form of regular expressions, and then print all lines in the file that matched that regular expression.

We thought that we'd like to generalize the class of patterns to deal with numbers as well as strings. We also thought that we'd like to have more computational capability than just printing the line that matched the pattern.

So out of this grew AWK, a language based on the principle of pattern-action processing. It was built to do simple data processing: the ordinary data processing that we routinely did on a day-to-day basis. We just wanted to have a very simple scripting language that would allow us, and people who weren't very computer savvy, to be able to write throw-away programs for routine data processing.

Were there any programs or languages that already had these functions at the time you developed AWK?

Our original model was GREP. But GREP had a very limited form of pattern action processing, so we generalized the capabilities of GREP considerably. I was also interested at that time in string pattern matching algorithms and context-free grammar parsing algorithms for compiler applications. This means that you can see a certain similarity between what AWK does and what the compiler construction tools LEX and YACC do.

LEX and YACC were tools that were built around string pattern matching algorithms that I was working on: LEX was designed to do lexical analysis and YACC syntax analysis. These tools were compiler construction utilities which were widely used in Bell labs, and later elsewhere, to create all sorts of little languages. Brian Kernighan was using them to make languages for typesetting mathematics and picture processing.

LEX is a tool that looks for lexemes in input text. Lexemes are sequences of characters that make up logical units. For example, a keyword like `then` in a programming language is a lexeme. The character `t` by itself isn't interesting, `h` by itself isn't interesting, but the combination `then` is interesting. One of the first tasks a compiler has to do is read the source program and group its characters into lexemes.

AWK was influenced by this kind of textual processing, but AWK was aimed at data-processing tasks and it assumed very little background on the part of the user in terms of

programming sophistication.

Can you provide Computerworld readers with a brief summary in your own words of AWK as a language?

AWK is a language for processing files of text. A file is treated as a sequence of records, and by default each line is a record. Each line is broken up into a sequence of fields, so we can think of the first word in a line as the first field, the second word as the second field, and so on. An AWK program is a sequence of pattern-action statements. AWK reads the input a line at a time. A line is scanned for each pattern in the program, and for each pattern that matches, the associated action is executed.

A simple example should make this clear. Suppose we have a file in which each line is a name followed by a phone number. Let's say the file contains the line `Naomi 1234`. In the AWK program the first field is referred to as `$1`, the second field as `$2`, and so on. Thus, we can create an AWK program to retrieve Naomi's phone number by simply writing `$1 == "Naomi" {print $2}` which means if the first field matches `Naomi`, then print the second field. Now you're an AWK programmer! If you typed that program into AWK and presented it with the file that had names and phone numbers, then it would print `1234` as Naomi's phone number.

A typical AWK program would have several pattern-action statements. The patterns can be Boolean combinations of strings and numbers; the actions can be statements in a C-like programming language.

AWK became popular since it was one of the standard programs that came with every Unix system.

What are you most proud of in the development of AWK?

AWK was developed by three people: me, Brian Kernighan and Peter Weinberger. Peter Weinberger was interested in what Brian and I were doing right from the start. We had created a grammatical specification for AWK but hadn't yet created the full run-time environment. Weinberger came along and said 'hey, this looks like a language I could use myself,' and within a week he created a working run time for AWK. This initial form of AWK was very useful for writing the data processing routines that we were all interested in but more importantly it provided an evolvable platform for the language.

One of the most interesting parts of this project for me was that I got to know how Kernighan and Weinberger thought about language design: it was a really enlightening process! With the flexible compiler construction tools we had at our disposal, we very quickly evolved the language to adopt new useful syntactic and semantic constructs. We spent a whole year intensely debating what constructs should and shouldn't be in the language.

Language design is a very personal activity and each person brings to a language the classes of problems that they'd like to solve, and the manner in which they'd like them to be solved. I had a lot of fun creating AWK, and working with Kernighan and Weinberger was one of the most stimulating experiences of my career. I also learned I would not want to get into a programming contest with either of them however! Their programming abilities are formidable.

Interestingly, we did not intend the language to be used except by the three of us. But very quickly we discovered lots of other people had the need for the routine kind of data processing that AWK was good for. People didn't want to write hundred-line C programs to do data processing that could be done with a few lines of AWK, so lots of people started using AWK.

For many years AWK was one of the most popular commands on Unix, and today, even though a number of other similar languages have come on the scene, AWK still ranks among the top 25 or 30 most popular programming languages in the world. And it all began as a little exercise to create a utility that the three of us would find useful for our own use.

How do you feel about AWK being so popular?

I am very happy that other people have found AWK useful. And not only did AWK attract a lot of users, other language designers later used it as a model for developing more powerful languages.

About 10 years after AWK was created, Larry Wall created a language called Perl, which

was patterned after AWK and some other Unix commands. PERL is now one of the most popular programming language in the world. So not only was AWK popular when it was introduced but it also stimulated the creation of other popular languages.

AWK has inspired many other languages as you've already mentioned: why do you think this is?

What made AWK popular initially was its simplicity and the kinds of tasks it was built to do. It has a very simple programming model. The idea of pattern-action programming is very natural for people. We also made the language compatible with pipes in Unix. The actions in AWK are really simple forms of C programs. You can write a simple action like `{print $2}` or you can write a much more complex C-like program as an action associated with a pattern. Some Wall Street financial houses used AWK when it first came out to balance their books because it was so easy to write data-processing programs in AWK.

AWK turned a number of people into programmers because the learning curve for the language was very shallow. Even today a large number of people continue to use AWK, saying languages such as PERL have become too complicated. Some say PERL has become such a complex language that it's become almost impossible to understand the programs once they've been written.

Another advantage of AWK is that the language is stable. We haven't changed it since the mid 1980s. And there are also lots of other people who've implemented versions of AWK on different platforms such as Windows.

How did you determine the order of initials in AWK?

This was not our choice. When our research colleagues saw the three of us in one or another's office, they'd walk by the open door and say 'AWK! AWK!.' So, we called the language AWK because of the good natured ribbing we received from our colleagues. We also thought it was a great name, and we put the auk bird picture on the AWK book when we published it.

What did you learn from developing AWK that you still apply in your work today?

My research specialties include algorithms and programming languages. Many more people know me for AWK as they've used it personally. Fewer people know me for my theoretical papers even though they may be using the algorithms in them that have been implemented in various tools. One of the nice things about AWK is that it incorporates efficient string pattern matching algorithms that I was working on at the time we developed AWK. These pattern matching algorithms are also found in other Unix utilities such as EGREP and FGREP, two string-matching tools I had written when I was experimenting with string pattern matching algorithms.

What AWK represents is a beautiful marriage of theory and practice. The best engineering is often built on top of a sound scientific foundation. In AWK we have taken expressive notations and efficient algorithms founded in computer science and engineered them to run well in practice.

I feel you gain wisdom by working with great people. Brian Kernighan is a master of useful programming language design. His basic precept of language design is to keep a language simple, so that a language is easy to understand and easy to use. I think this is great advice for any language designer.

Have you had any surprises in the way that AWK has developed over the years?

One Monday morning I walked into my office to find a person from the Bell Labs micro-electronics product division who had used AWK to create a multi-thousand-line computer-aided design system. I was just stunned. I thought that no one would ever write an AWK program with more than handful of statements. But he had written a powerful CAD development system in AWK because he could do it so quickly and with such facility. My biggest surprise is that AWK has been used in many different applications that none of us had initially envisaged. But perhaps that's the sign of a good tool, as you use a screwdriver for many more things than turning screws.

Do you still work with AWK today?

Since it's so useful for routine data processing I use it daily. For example, I use it whenever I'm writing papers and books. Because it has associative arrays, I have a simple two-line AWK program that translates symbolically named figures and examples into numerically encoded

figures and examples; for instance, it translates `Figure AWK-program` into `Figure 1.1`. This AWK program allows me to rearrange and renumber figures and examples at will in my papers and books. I once saw a paper that had a 1000-line C that had less functionality than these two lines of AWK. The economy of expression you can get from AWK can be very impressive.

How has being one of the three creators of AWK impacted your career?

As I said, many programmers know me for AWK, but the computer science research community is much more familiar with my theoretical work. So I initially viewed the creation of AWK as a learning experience and a diversion rather than part of my regular research activities. However, the experience of implementing AWK has greatly influenced how I now teach programming languages and compilers, and software engineering.

What I've noticed is that some scientists aren't as well known for their primary field of research by the world at large as they are for their useful tools. Don Knuth, for example, is one of the world's foremost computer scientists, a founder of the field of computer algorithms. However, he developed a language for typesetting technical papers, called TEX. This wasn't his main avenue of research but TEX became very widely used throughout the world by many scientists outside of computer science. Knuth was passionate about having a mathematical typesetting system that could be used to produce beautiful looking papers and books.

Many other computer science researchers have developed useful programming languages as a by-product of their main line of research as well. As another example, Bjarne Stroustrup developed the widely used C++ programming language because he wanted to write network simulators.

Would you do anything differently in the development of AWK looking back?

One of the things that I would have done differently is instituting rigorous testing as we started to develop the language. We initially created AWK as a throw-away language, so we didn't do rigorous quality control as part of our initial implementation.

I mentioned to you earlier that there was a person who wrote a CAD system in AWK. The reason he initially came to see me was to report a bug in the AWK complier. He was very testy with me saying I had wasted three weeks of his life, as he had been looking for a bug in his own code only to discover that it was a bug in the AWK compiler! I huddled with Brian Kernighan after this, and we agreed we really need to do something differently in terms of quality control. So we instituted a rigorous regression test for all of the features of AWK. Any of the three of us who put in a new feature into the language from then on, first had to write a test for the new feature.

I have been teaching the programming languages and compilers course at Columbia University, for many several years. The course has a semester long project in which students work in teams of four or five to design their own innovative little language and to make a compiler for it.

Students coming into the course have never looked inside a compiler before, but in all the years I've been teaching this course, never has a team failed to deliver a working compiler at the end of the course. All of this is due to the experience I had in developing AWK with Kernighan and Weinberger. In addition to learning the principles of language and compiler design, the students learn good software engineering practices. Rigorous testing is something students do from the start. The students also learn the elements of project management, teamwork, and communication skills, both oral and written. So from that perspective AWK has significantly influenced how I teach programming languages and compilers and software development.

Bash: Chet Ramey

Bash, or the Bourne-Again Shell is a Unix shell created in 1987 by Brian Fox. According to Wikipedia, the name is a pun on an earlier Unix shell by Stephen Bourne (called the Bourne shell), which was distributed with Version 7 Unix in 1978.
In 1990, Chet Ramey, Manager of the Network Engineering and Security Group in Technology Infrastructure Services at Case Western Reserve University, became the primary maintainer of the language.
Computerworld tracked down Ramey to find out more

How did you first become involved with Bash?

In 1989 or so, I was doing network services and server support for [Case Western Reserve] University (CWRU), and was not satisfied with the shells I had available for that work. I wasn't really interested in using SH for programming and CSH/TCSH for interactive use, so I began looking around for a version of SH with the interactive features I wanted (job control, line editing, command history, filename completion, and so on.)

I found a couple of versions of the SVR2 shell where those features had been added (credit to Doug Gwyn, Ron Natalie, and Arnold Robbins, who had done the work). These were available to CWRU because we were Unix source licensees, but I had trouble with them and couldn't extend them the way I wanted. Ken Almquist was writing ASH, but that had not been released, and there was a clone of the 7th edition shell, which eventually became *PDksh*, but that did not have the features I wanted either.

Brian Fox had begun writing BASH and readline (which was not, at that time, a separate library) the year before, when he was an employee of the FSF. The story, as I recall it, was that a volunteer had come forward and offered to write a Bourne Shell clone. After some time, he had produced nothing, so Richard Stallman directed Brian to write a shell. Stallman said it should take only a couple of months.

I started looking again, and ended up finding a very early version of BASH. I forget where I got it, but it was after Brian had sent a copy to Paul Placeway from Ohio State – Paul had been the TCSH maintainer for many years, and Brian asked him to help with the line editing and redisplay code. I took that version, made job control work and fixed a number of other bugs, and sent my changes to Brian. He was impressed enough to begin working with me, and we went on from there.

I fixed many of the bugs people reported in the first public versions of BASH and fed those fixes back to Brian. We began working together as more or less co-maintainers, and when Brian moved on to other things, I still needed to support BASH for my local users, so I produced several local releases. Brian and I eventually merged those versions together, and when he moved away from BASH development, I took over.

Did you work with Brian Fox before becoming the primary maintainer of the language?

Brian and I worked together for several years before he moved on to other things. The versions through bash-1.13 were collaborative releases.

What is/was your working relationship with Brian like?

Our working relationship was very good, especially considering we met in person only once, in 1990. We were heavy users of Unix *talk* and *ntalk*, which allowed real-time two-way communication over the Internet back then, and made good use of email and the occasional long distance phone call. We still stay in touch.

What prompted the making of Bash in the first place?

When Richard Stallman decided to create a full replacement for the then-encumbered Unix systems, he knew that he would eventually have to have replacements for all of the common utilities, especially the standard shell, and those replacements would have to have acceptable licensing. After a couple of false starts (as previously mentioned), he hired Brian Fox to write

it. They decided early on that they would implement the shell as defined by the Posix standard, and used that as a specification.

Was there a particular problem that the language aimed to solve?

In BASH's case, the problem to be solved was a free software version of the Posix standard shell to be part of the GNU system.

The original version of the shell (Steve Bourne's version) was intended to overcome a number of the limitations of the Unix shell included in versions up to the sixth edition, originally written by Ken Thompson.

Why did you take over as the language's primary maintainer three years after Fox created the language?

Brian wanted to move on to other things, and I was a developer willing to take it on and experienced with the code. Brian and the FSF trusted me with the program's future.

What prompted the writing of the GNU Bash Reference Manual and the Bash Reference Manual?

Any good heavily-used program needs good reference documentation, and BASH is no exception. I originally wrote the documents to support my local users, and they were folded into official releases along the line.

Is there a strong relationship between the original Bourne Shell and the Bourne-Again Shell?

I'd say there is a linear relationship: the original Bourne Shell was very influential, the various System V shell releases preserved that heritage, and the Posix committee used those versions as the basis for the standard they developed.

Certainly the basic language syntax and built-in commands are direct descendants of the Bourne shell's. BASH's additional features and functionality build on what the Bourne shell provided. As for source code and internal implementation, there's no relationship at all, of course.

What prompted the language's name: why was a pun created on the Bourne Shell?

The FSF has a penchant for puns, and this one seemed appropriate, I suppose. The name predates my involvement.

Have you faced any hard decisions in maintaining the language?

The hardest decisions are the ones dealing with compatibility: how compatible to be with the versions of SH existing at various points throughout BASH's history; how compatible to be with the features from the Korn shell I considered valuable; where and how to differ from the Posix standard, and when to break backwards compatibility with previous BASH versions to correct mistakes I had made.

Some of the features implemented (and not implemented) required a lot of thought and consideration – not how to implement them, but whether or not to invest the resources to do so. Most of the BASH development over the past 15 years has been done by one person.

Are you still working with the language now?

I am. In fact, the next major release of BASH, bash-4.0, should be out sometime this (Northern) summer.

What is the latest project you have used it for?

I mostly use BASH for interactive work these days. I use it to write some small system administration tools, but I don't do much system administration any more.

What is the most exciting piece of code (that you know of) ever written in Bash?

That's hard to say. Lots of interesting projects have been implemented as shell scripts, or sets of shell scripts.

I particularly like the various versions of the BASH debugger that were implemented completely as shell scripts. That's pretty complex work. I've seen entire Web servers and other surprisingly substantial applications written as shell scripts.

In your opinion, what lasting legacy has Bash brought to the Web?

I think BASH's legacy is as a solid piece of infrastructure, and the shell making millions of Linux, Mac OS X, and Solaris systems work every day.

As I recall, it was one of the first couple of programs Linus Torvalds made run on his early Linux kernels.

Where do you envisage Bash's future lying?

BASH will continue to evolve as both an interactive environment and a programming language. I'd like to add more features that allow interested users to extend the shell in novel ways. The programmable completion system is an example of that kind of extension.

BASH's evolution has always been user-driven, so it will ultimately be up to the feature requests that come in.

Where do you see computer programming languages heading in the future, particularly in the next five to 20 years?

I see increased dynamism, allowing programmers to do more and more complex things on the fly, especially over the Web. The advances in hardware allow interpreted code to run faster today than compiled code on some systems available when I started work on BASH.

Do you have any advice for up-and-coming programmers?

Find an area that interests you and get involved with an existing community. There are free software projects in just about any area of programming.

The nuts-and-bolts – which language you use, what programming environment you use, where you do your work – are not as important as the passion and interest you bring to the work itself.

Is there anything else that you'd like to add?

The free software community is still as vibrant today, maybe even more so, than when I first became involved. There is still a lot of room for significant contributions; all it takes is an interested person with a good idea.

C#: Anders Hejlsberg

Microsoft's leader of C# development, Anders Hejlsberg, took some time to tell Computerworld about the development of C#, his thoughts on future programming trends, and his experiences putting out fires. Hejlsberg is also responsible for writing the Turbo Pascal system, and was the lead architect on the team that developed Delphi

What were the fundamental flaws in other languages that you believe drove the development of Common Language Runtime (CLR), and in turn, C#?

I wouldn't say that our primary motivation for CLR was fundamental flaws in other languages. But we certainly had some key goals in mind. Primarily, we wanted to build a unified and modern development platform for multiple programming languages and application models.

To put this aim in context, we need to look back to the inception of .NET, which was in the late nineties or early 2000s. At that time, Microsoft's primary developer offerings were fairly fragmented. For native code we had C++ with MFC, and ATL and so forth. And then for rapid application development we had Visual Basic, and for Web development we had IIS and ASP.

Each language was its own little silo with different solutions to all of the different programming problems. You couldn't transfer your skills and your application model implicitly became your choice of programming language. We really wanted to unify these separate entities to better leverage our efforts.

We also wanted to introduce modern concepts, such as object orientation, type safety, garbage collection and structured exception handling directly into the platform. At the time, the underlying infrastructure we were running on was COM, which is a very low-level programming model that requires you to deal with the registry and reference counting and HRESULTs and all that stuff.

These factors were, at the time, the motivators for .NET. There was also a competitive angle with Sun and Java etc.

Now, to move on to C#, in a nutshell our aim was to create a first class modern language on this platform that would appeal to the curly braces crowd: the C++ programmers of the world at the time, and competitively, the Java programmers.

There were several elements that we considered key design goals, like support for the next level up from object-oriented programming, to component-based programming where properties and metadata attributes were all first class in the language. Also, a unified and extensible type system, which sort of gets into value types and boxing etc. Versioning was a big thing; making sure we designed the language so that it would version well, so that whenever we added new features to the language we would not break code in older applications. These were all values that were important to us.

Of course, at the end of the day, productivity has always been a driver for me in all of the projects I've worked on. It's about making programmers more productive.

Why was the language originally named Cool, and what promoted the change to C#?

The code name was Cool, which stood for 'C-like Object Oriented Language.' We kind of liked that name: all of our files were called `.cool` and that was kind of cool! We looked seriously at keeping the name for the final product but it was just not feasible from a trademark perspective, as there were way too many cool things out there.

So the naming committee had to get to work and we sort of liked the notion of having an inherent reference to C in there, and a little word play on C++, as you can sort of view the sharp sign as four pluses, so it's C++++. And the musical aspect was interesting too. So C# it was, and I've actually been really happy with that name. It's served us well.

How has your experience designing Visual J++, Borland Delphi and Turbo Pascal impacted on C#?

If you go back to the Turbo Pascal days, the really new element created by Turbo Pascal was

that it was the first product ever to commercialize the integrated development environment, in a broad sense – the rapid turnaround cycle between compile, edit or edit, compile, debug. Any development tool today looks that same way, and that of course has always been a key thing.

[I also learnt to] design the language to be well-toolable. This does impact the language in subtle ways – you've got to make sure the syntax works well for having a background compiler, and statement completion. There are actually some languages, such as SQL, where it's very hard to do meaningful statement completion as things sort of come in the wrong order. When you write your `SELECT` clause, you can't tell what people are selecting from, or what they might select until after writing the `FROM` clause. There are things like that to keep in mind.

Each of the products I've worked on, I'd say, have taught me valuable lessons about what works and what doesn't, and of course you end up applying that knowledge to subsequent products you work on. For example, Delphi was the first product I worked on to natively support properties, and then that got carried over to C# for example. We added a similar feature there.

Have you encountered any major problems in the development of C#? Any catastrophes?

No, I wouldn't say that there have been any catastrophes! But life is nothing but little missteps and corrections along the way, so there are always little fires you're putting out, but I wouldn't say we ever had total meltdowns. It's been a lot of fun to work on and it's been over 10 years now.

Can you give me an example of a little fire that you've had to put out?

Every project is about not what you put in, but what you don't have time to put in! So it's always about what we're going to cut ... so every project is like that. It's so hard to single out anything in particular as we're always putting out fires. New people leave the team and new people come in, it's like every day you come to work and there's something new to be dealt with.

Would you do anything differently in developing C# if you had the chance?

There are several things. First of all, when we shipped C# 1.0 we did not have generics in the language – that came in C# 2.0, and the minute we shipped generics we were able to put a lot of old code to bed as it was superfluous and not as strongly typed as generics. So a bunch of stuff got deprecated right out of the box in C#2.0. We knew generics were coming but it was one of those hard decisions: do you hold the platform longer or do you ship now and work on this and then ship it a couple of years later? I would have loved to have generics from the beginning as it would have left us with less obsolete stuff in the framework today.

With language design or with platform design 1.0 is always a unique opportunity to put down your core values, your core designs, and then with every version thereafter it's much harder to fundamentally change the nature of the beast. And so, the things that you typically end up regretting later are the fundamentals that you didn't quite get right. Because those you can't change – you can always ship new libraries etc, but you can't change the fundamental gestalt of the platform.

For example, in the type system we do not have separation between value and reference types and nullability of types. This may sound a little wonky or a little technical, but in C# reference types can be null, such as strings, but value types cannot be null. It sure would be nice to have had non-nullable reference types, so you could declare that 'this string can never be null, and I want you compiler to check that I can never hit a null pointer here.'

50% of the bugs that people run into today, coding with C# in our platform, and the same is true of Java for that matter, are probably null reference exceptions. If we had had a stronger type system that would allow you to say that 'this parameter may never be null, and you compiler please check that at every call, by doing static analysis of the code.' Then we could have stamped out classes of bugs.

But peppering that on after the fact once you've built a whole platform where this isn't built in ... it's very hard to pepper on afterwards. Because if you start strengthening your APIs and saying that you can't pass null here or null here or null here, then all of a sudden you're starting

to break a bunch of code. It may not be possible for the compiler to track it all properly.

Anyway, those are just things that are tough later. You sort of end up going, well ok, if we ever get another chance in umpteen years to build a new platform, we'll definitely get this one right. Of course then we'll go and make other mistakes! But we won't make that one.

Why do you think C is such a popular language base, with many languages built on it such as C++ and C#?

I think you have to take the historic view there first. If you go back to C itself, C was a very, very appropriate language for its time. It was really the language that lifted operating system builders out of assembly code and gave them higher-level abstractions such as data types and so forth, yet was sufficiently close to the machine so that you could write efficient code. It was also very succinct: it was a very terse language, you can write very compact code which is something that programmers very much prefer. You compare a C program to a COBOL program and I can tell you where you're going to see more characters.

So C was just an incredibly appropriate language for its time, and C++ was an incredibly appropriate evolution of C. Once you have huge language use, it is much easier to evolve and bring an existing base with you than it is to go create something brand new. If you look at the mechanics of new languages, when you design a new language you can either decide to evolve an existing language or start from scratch.

Evolving an existing language means you have an instantaneous big user base, and everything you add to the language is just gravy ... there's really no drawback as all of the old code still works. Start with a brand new language and you essentially start with minus 1,000 points. And now, you've got to win back your 1,000 points before we're even talking. Lots of languages never get to more than minus 500. Yeah, they add value but they didn't add enough value over what was there before. So C++ I think is a fantastic example of a very appropriate evolution of an existing language. It came right at the dawn of object-oriented programming and pioneered that right into the core programming community, in a great way.

Of course by the time we started looking at C# as a new language, there was a huge, huge number of programmers out there that were very accustomed to programming with curly braces, like the C guys, C++ guys, Java guys etc etc. And so for us that was a very natural starting point: to make a language that would appeal to C++ programmers and to Java programmers. And that really meant build a language in the C heritage. And I think that has served us very, very well.

What do you think of the upcoming language F#, which is touted as a fusion of a functional language and C#?

I'm very enthusiastic about F# and the work that Don Syme from Microsoft Research in Cambridge is doing on this language. I wouldn't say it's a fusion of ML and C#. I mean, certainly its roots come from the ML base of functional programming languages, and it is closely related to Caml. I view it as a fusion of Caml and .NET, and a great impact of tooling experience.

Do you think that it's ever going to become a large competitor to C#?

I think they are both great and very complementary. A competitor, yes, in the sense that VB is a competitor. But do you think of them as competitors? Or do you think of them as languages on a unified platform? I mean, I don't personally: to me, the important thing is what's built on top of .NET. Every language borrows from other languages, but that's how we make progress in the industry and I'm interested in progress.

What do you think of functional programming in general?

I think that functional programming is an incredibly interesting paradigm for us to look at, and certainly if you look at C# 3.0, functional programming has been a primary inspiration there, in all that we've done with LINQ and all of the primitive language features that it breaks down to. I think the time has finally come for functional programming to enter the mainstream. But, mainstream is different from taking over the world.

I definitely think that there is a space for functional programming today, and F# is unique in being the first industrial strength functional programming language with an industrial strength

tooling language behind it, and an industrial strength platform underneath it. The thing that's really unique about F# compared to all of the other functional programming languages is that it really offers first class support for object-oriented programming as well, and first class interoperability with the .NET framework. Anything we have in the .NET framework is as easy to use from F# as it is from C# as it is from VB – it does not feel forced.

A lot of functional programming languages have lived in their own little world, and they've been pure and mathematical and so forth, but you couldn't get to the big library that's out there. If you look at languages today, they live and die by whether they have good framework support, as the frameworks are so big and so huge and so rich that you just cannot afford to ignore them anymore. And that's why you're seeing so many languages being built on top of .NET or on top of Java as opposed to being built in their own little worlds.

How do you feel about C# becoming standardized and adopted by Microsoft?

If you're asking from a personal perspective, I think it's fantastic. I've been super fortunate to have Microsoft give me the opportunity to be the chief architect of a programming language and then have the company put its might behind it. That's not an opportunity you get every day, and it's been great.

With respect to standardization, I have always been a strong supporter of standardizing the language and I have always felt that you can't have your cake and eat it too when it comes to expecting a language to be proprietary and also wanting community investment in the language. Be proprietary, but then just don't expect people to build stuff on top of it. Or, you can open it up and people will feel more comfortable about investing.

Now, you can argue that we're not obviously open source or anything, but the language is standardized, and the entire specification is available for anyone to go replicate. Mono has done so, and I think Mono is a fantastic thing. I don't know [if] you're familiar with Mono, but it's an implementation of the C# standard and the CLI standard (which is effectively the .NET standard) on Linux, built as an open source project. And they're doing great work and we talk to them a lot and I think it's a super thing.

And I guess they couldn't have done that had you not put the specifications out there?

Well, they could have but it would have been a heck of a lot harder and it would probably not be as good a product. You can go reverse engineer it ... they have reverse engineered a lot of the .NET platform ... but all of the core semantics of the language, they were part of the standardization process.

You know most recently we've created Silverlight, which is our browser hosted .NET runtime environment, and the Mono guys have built a project called Moonlight which is an implementation of Silverlight that is officially sanctioned by Microsoft that runs on Linux and other browsers. It's a good thing.

So to focus more specifically on C#, why did you decide to introduce boxing & unboxing into the language?

I may have even touched on that a little bit earlier. What it boils down to is the fact that boxing allows you to unify the type system, and what I mean by unify is that when you are learning C# or approaching the language for the first time, you can make the simple statement that 'in this language, everything is an object.' Any piece of data you have you can treat as an object and assign it to a variable type object. The mechanism that makes that work is boxing and unboxing.

If you look at a similar language such as Java, it has a divided type system where everything is an object except ints and bools and characters etc which are not objects. So you have to sort of immediately dive in and describe the finer distinctions between these classes and types. Whereas when you have a unified type system you can just treat them as objects and then later, if you care, you can start diving into the deeper details about value types vs. reference types and what the mechanics are about and so forth.

We've seen this many times as people that teach the language have come back and said this is great as it allows us to have a very simple starting point. So from a pedagogical standpoint,

it flows much better to first say that everything is an object and later we'll teach you about the different kinds of objects that the system has.

Did you intend to make it easy to teach, or was that simply a side effect of the way the language was designed?

I'd say we kept teachability in mind. It's not just teachability that is an advantage of a unified type system, but it also allows your programs to have fewer special cases etc. I would say the motivator here was more conceptual simplicity. But conceptual simplicity is generally a great thing when it comes to teachability so the two kind of go hand in hand.

How do you feel about C# 3.0? Were you happy with the release? When is the next release due out?

Yes, I'm very happy with it, I think in some ways C# 3.0 was our first chance to truly do innovation and something brand new in the language. C# 1.0, if you think about it, was like 'let's go from zero to somewhere, so lets build all of the core things that a programming language has to have.' So, in a sense, 'let's build the 90% that is already known in the world out there.' C# 2.0 was about doing all of the things we wanted to do in C# 1.0 but we knew we weren't going to have time to do. So C# 3.0 was the first chance of a green field: ok, what big problem are we going to attack here?

The problem we chose to attack was the mismatch between databases and general purpose programming languages, and the lack of queries and more declarative styles of programming in general purpose programming languages. It was a fantastic voyage, and it was so much fun to work on. The result has been quite unique and quite good really. LINQ is something that is a new thing.

Do you expect C#3.0 to become an ECMA and ISO standard, as previous versions have?

We're certainly open to that. There's no ongoing work in the standards committee at the moment, but it's really more a question of whether the community of industry partners out there would like to continue with that process. I should also say that the standards for C# explicitly do permit implementers to have extensions to the language, so though C# 3.0 is not standardized, it is certainly a complete implementation of the C# 2.0 standard. It is 100% backwards compatible, as all versions are.

What functionality do you hope to add to C# in the future versions?

There are many. I have a huge laundry list, or our team does, of features that people have requested over the years. If I had to name the 3 big trends that are going on in the industry that we take an interest in and get inspiration from, I would say the first is a move towards more declarative styles of programming, and you can sort of see LINQ as an example of that. All the talk we have about domain specific languages, that's one form of declarative programming, and functional programming is another style of declarative programming. I think those are going to be quite important going forward and are certainly areas that we will invest in, in C#.

Dynamic programming is seeing a big resurgence these days, if you look at phenomena like Ruby and Ruby on Rails, these are all of a sudden very popular, and there are certain things you can do with dynamic programming languages that it would be great to also have in more classical languages like C#. So that's something we're also looking at.

Lastly, I would say that concurrency is the big thing that you can't ignore these days because the mechanics of Moore's law are such that it is no longer feasible to build more powerful processors. We can't make them faster anymore because we can't get rid of the heat, and so now all the acreage on the chips is being used to make more processors and all of a sudden it's almost impossible to get a machine that doesn't have multiple CPUs.

Right now you might have two cores but it's only a matter of years before you have 4 or 8 or more than that, even in a standard desktop machine. In order for us to take advantage of that, we need much better programming models for concurrency. That's a tough problem, it's a problem that doesn't just face us but the entire industry, and lots of people are thinking about it and we certainly are amongst those.

There's no shortage of problems to solve!

Speaking of problems, how do you respond to criticism of C#, such as that the .NET platform only allows the language to run on Windows, as well as licensing and performance concerns?

It is possible to build alternate implementations. We are not building .NET for Linux, because the value proposition that we can deliver to our customers is a complete unified and thoroughly tested package, from the OS framework to databases to Web servers etc. So .NET is part of a greater ecosystem, and all of these things work together. I think we are actually running on certain other platforms, such as Mono on Linux and other third party implementations. Silverlight now allows you to run .NET applications inside the browser and not just in our browser, but also in Safari on Macs for example.

As for performance concerns, I feel very comfortable about .NET performance compared to competitive platforms. I feel very good about it actually. There are performance issues here and there, as there is with anything, but I feel like we are always on a vigilant quest to make performance better and performance is pretty darn good. Performance is one of the key reasons that people choose .NET, certainly in the case studies I see and the customers I talk to (productivity being the other.)

What's the most unusual/interesting program you've ever seen written in C#?

Microsoft Research has this really cool application called Worldwide Telescope, which is written in C#. It's effectively a beautiful interface on a catalogue of astronomical images (or images from astronomy) which allow you to do infinite zooming in on a planet and to see more and more detail. If you happen to choose planet Earth you can literally zoom in from galactic scale to your house, which is cool. I've been playing around with it with my kids and looking at other planets and they think it's fun. It popularizes a thing that has traditionally been hard to get excited about.

Do you always use the Visual C# compiler, or do you ever use versions developed by the Mono or DotGNU projects?

I day to day use Visual Studio and Visual C# as that's the environment I live in. I occasionally check out the Mono project or some of the other projects, but that's more intellectual curiosity, rather than my day to day tool.

In your opinion, what lasting legacy has C# brought to Computer development?

We all stand on the shoulders of giants here and every language builds on what went before it so we owe a lot to C, C++, Java, Delphi, all of these other things that came before us ... we now hope to deliver our own incremental value.

I would say I'm very happy that C# definitely brought a big productivity boost to developers on the Windows platform and we continue to see that.

I think that C# is becoming one of the first widely adopted multi-paradigm programming languages out there. With C# you can do object-oriented programming, you can do procedural programming, now you can also do functional programming with a bunch of the extensions we've added in C# 3.0. We're looking at C# 4.0 supporting dynamic programming and so we aim to harvest the best from all of these previously distinct language categories and deliver it all in a single language.

In terms of specific contributions, I think the work we've done in C# 3.0 on language integrated queries certainly seems to be inspiring lots of other languages out there. I'm very happy with that and I'm certainly hoping that in 10 years there will be no languages where query isn't just an automatic feature: it will be a feature that you must have. So I think we've certainly advanced the state of the art there.

Has the popularity of the language surprised you at all?

It would have been presumptuous of me to say 'so today we're starting .NET and in 8 years we will own half of the world's development' or whatever. You can hope, but I have been pleasantly surprised.

Certainly we have labored hard to create a quality product, so it's nice to see that we're

being rewarded with lots of usage. At the end of the day, that's what keeps us going, knowing hundreds of thousands, if not millions of programmers use the stuff you work on day and you make their life better (hopefully!).

What are you working on now?

I'm always working on the next release, so you can add one and deduce we're working on C#4.0!

Do you have any idea when that release will be coming out?

I don't think we're saying officially now, but we're on a cadence of shipping every two years or so, or at least that's what we hope to do. So 2010 sometime hopefully ... there's a set of features that we're working on there that we're actually going to talk about at the PDC (Professional Developers Conference) at the end of October. We're giving some of the first presentations on what we're doing.

Where do you think programming languages will be heading in the future, particularly in the next 5 to 20 years?

I've been doing this now for 25 or almost 30 years, and I remember some early interviews that I gave after Turbo Pascal became very popular. People would always ask me where programming will be in 20 or so years (this is 1983 if you go back.) Of course, back then, the first thing out of one's mouth was well 'maybe we won't even be programming at all and maybe we'll actually just be telling computers what to do. If we're doing any programming at all it's likely to be visual and we'll just be moving around software ICs and drawing lines and boxes.'

Lo and behold here we are 25 years later. We're still programming in text and the programs look almost the same as they did 25 years ago. Yep, we've made a little bit of progress but it's a lot slower than everyone expected.

I'm going to be very cautious and not predict that we're going to be telling computers what to do, but that it will look a lot like it does today, but that we're going to be more productive, it's hopefully going to be more succinct, we're going to be able to say more with less code and we can be more declarative. We will hopefully have found good programming models for concurrency as that does seem to be an unavoidable trend.

Honestly, it's anyone's guess what it's going to look like in the next 20 years, but certainly in the next 5 years those are the things that are going to be keeping us busy.

And do you have any advice for up-and-coming programmers?

I think it's important to try to master the different paradigms of programs that are out there. The obvious object-oriented programming is hopefully something that you will be taught in school. Hopefully school will also teach you functional programming, if not, that is a good thing to go look at.

Go look at dynamic languages and meta-programming: those are really interesting concepts. Once you get an understanding of these different kinds of programming and the philosophies that underlie them, you can get a much more coherent picture of what's going on and the different styles of programming that might be more appropriate for you with what you're doing right now.

Anyone programming today should check out functional programming and meta-programming as they are very important trends going forward.

C++: Bjarne Stroustrup

Bjarne Stroustrup is currently the College of Engineering Chair and Computer Science Professor at Texas A&M University, and is an AT&T labs fellow. We chat to him about the design and development of C++, garbage collection and the role of facial hair in successful programming languages

What prompted the development of C++?

I needed a tool for designing and implementing a distributed version of the Unix kernel. At the time, 1979, no such tool existed. I needed something that could express the structure of a program, deal directly with hardware, and be sufficiently efficient and sufficiently portable for serious systems programming.

You can find more detailed information about the design and evolution of C++ in my HOPL (History of Programming Languages) papers, which you can find on my home pages (`http://www.research.att.com/~bs`), and in my book *The Design and Evolution of C++*.

Was there a particular problem you were trying to solve?

The two problems that stick in my mind were to simulate the inter-process communication infrastructure for a distributed or shared-memory system (to determine which OS services we could afford to run on separate processors), and [the need] to write the network drivers for such a system. Obviously – since Unix was written in C – I also wanted a high degree of C compatibility. Very early, 1980 onwards, it was used by other people (helped by me) for simulations of various network protocols and traffic management algorithms.

Where does the name C++ come from?

As *C with Classes* (my ancestor to C++) became popular within Bell Labs, some people found that name too much of a mouthful and started to call it C. This meant that they needed to qualify what they meant when they wanted to refer to Dennis Ritchie's language, so they used 'Old C,' 'Straight C,' and such. Somebody found that disrespectful to Dennis (neither Dennis nor I felt that) and one day I received a request though Bell Labs management channels to find a better name. As a result, we referred to C++ as C84 for a while. That didn't do much good, so I asked around for suggestions and picked C++ from the resulting list. Everybody agreed that semantically ++C would have been even better, but I thought that would create too many problems for non-geeks.

Were there any particularly difficult or frustrating problems you had to overcome in the development of the language?

Lots! For starters, what should be the fundamental design rules for the language? What should be in the language and what should be left out? Most people demand a tiny language providing every feature they have ever found useful in any language. Unfortunately, that's impossible.

After a short period of relying on luck and good taste, I settled on a set of rules of thumb intended to ensure that programs in C++ could be simultaneously elegant (as in Simula67, the language that introduced object-oriented programming) and efficient for systems programming (as in C). Obviously, not every program can be both and many are neither, but the intent was (and is) that a competent programmer should be able to express just about any idea directly and have it executed with minimal overheads (zero overheads compared to a C version).

Convincing the systems programming community of the value of type checking was surprisingly hard. The idea of checking function arguments against a function declaration was fiercely resisted by many – at least until C adopted the idea from C with Classes.

These days, object-oriented programming is just about everywhere, so it is hard for people to believe that I basically failed to convince people about it's utility until I finally just put in virtual functions and demonstrated that they were fast enough for demanding uses. C++'s variant of OOP was (and is) basically that of Simula with some simplifications and speedups.

C compatibility was (and is) a major source of both problems and strengths. By being C compatible, C++ programmers were guaranteed a completeness of features that is often missing

in first releases of new languages and direct (and efficient) access to a large amount of code – not just C code, but also FORTRAN code and more because the C calling conventions were simple and similar to what other languages supported. After all, I used to say, reuse starts by using something that already exists, rather than waiting for someone developing new components intended for reuse. On the other hand, C has many syntactic and semantic oddities and keeping in lockstep with C as it evolved has not been easy.

What are the main differences between the original C with Classes and C++?

Most of the differences were in the implementation technique. C with Classes was implemented by a preprocessor, whereas C++ requires a proper compiler (so I wrote one). It was easy to transcribe C with Classes programs into C++, but the languages were not 100% compatible. From a language point of view, the major improvement was the provision of virtual functions, which enabled classical object-oriented programming. Overloading (including operator overloading) was also added, supported by better support for inlining. It may be worth noting that the key C++ features for general resource management, constructors and destructors, were in the earliest version of C with Classes. On the other hand, templates (and exceptions) were introduced in a slightly later version of C++ (1989); before that, we primarily used macros to express generic programming ideas.

Would you have done anything differently in the development of C++ if you had the chance?

This common question is a bit unfair because of course I didn't have the benefits of almost 30 years of experience with C++ then, and much of what I know now is the result of experimentation with the earlier versions of C++. Also, I had essentially no resources then (just me – part time) so if I grandly suggest (correctly) that virtual functions, templates (with *concepts* similar to what C++0x offers), and exceptions would have made C++85 a much better language, I would be suggesting not just something that I didn't know how to design in the early 1980s but also something that – if I magically had discovered the perfect design – couldn't have been implemented in a reasonable time.

I think that shipping a better standard library with C++ 1.0 in 1985 would have been barely feasible and would have been the most significant improvement for the time. By a 'better library' I mean one with a library of foundation classes that included a slightly improved version of the (then available and shipping) task library for the support of concurrency and a set of container classes. Shipping those would have encouraged development of improved versions and established a culture of using standard foundation libraries rather than corporate ones.

Later, I would have developed templates (key to C++ style generic programming) before multiple inheritance (not as major a feature as some people seem to consider it) and emphasized exceptions more. However, 'exceptions' again brings to a head the problem of hindsight. Some of the most important concepts underlying the modern use of templates on C++ did not exist until a bit later. For example the use of guarantees in describing safe and systematic uses of templates was only developed during the standardization of C++, notably by Dave Abrahams.

How did you feel about C++ becoming standardized in 1998 and how were you involved with the standardization process?

I worked hard on that standard for years (1989-1997) – as I am now working on its successor standard: C++0x. Keeping a main-stream language from fragmenting into feuding dialects is a hard and essential task. C++ has no owner or 'sugar daddy' to supply development muscle, free libraries, and marketing. The ISO standard committee was essential for the growth of the C++ community and that community owes an enormous amount to the many volunteers who worked (and work) on the committee.

What is the most interesting program that you've seen written with C++?

I can't pick one and I don't usually think of a program as interesting. I look more at complete systems – of which parts are written in C++. Among such systems, NASA's Mars Rovers' autonomous driving sub-system, the Google search engine, and Amadeus' airline reservation system spring to mind. Looking at code in isolation, I think Alexander Stepanov's STL (the

containers, iterators, and algorithms part of the C++ standard library) is among the most interesting, useful, and influential pieces of C++ code I have ever seen.

Have you ever seen the language used in a way that was not originally intended?

I designed C++ for generality. That is, the features were deliberately designed to do things I couldn't possibly imagine – as opposed to enforce my views of what is good. In addition, the C++ abstraction facilities (e. g., classes and templates) were designed to be optimally fast when used on conventional hardware so that people could afford to build the basic abstractions they need for a given application area (such as complex numbers and resource handles) within the language.

So, yes, I see C++ used for many things that I had not predicted and used in many ways that I had not anticipated, but usually I'm not completely stunned. I expected to be surprised, I designed for it. For example, I was very surprised by the structure of the STL and the look of code using it – I thought I knew what good container uses looked like. However, I had designed templates to preserve and use type information at compile time and worked hard to ensure that a simple function such as less-than could be inlined and compiled down to a single machine instruction. That allowed the weaving of separately defined code into efficient executable code, which is key to the efficiency of the STL. The biggest surprise, I guess, was that the STL matched all but one of a long list of design criteria for a general purpose container architecture that I had compiled over the years, but the way STL code looked was entirely unexpected.

So I'm often pleased with the surprises, but many times I'm dismayed at the attempts to force C++ into a mold for which it is not suited because someone didn't bother to learn the basics of C++. Of course, people doing that don't believe that they are acting irrationally; rather, they think that they know how to program and that there is nothing new or different about C++ that requires them to change their habits and learn new tricks. People who are confident in that way structure the code exactly as they would for, say, C or Java and are surprised when C++ doesn't do what they expect. Some people are even angry, though I don't see why someone should be angry to find that they need to be more careful with the type system in C++ than in C or that there is no company supplying free and standard libraries for C++ as for Java. To use C++ well, you have to use the type system and you have to seek out or build libraries. Trying to build applications directly on the bare language or with just the standard library is wasteful of your time and effort. Fighting the type system (with lots of casts and macros) is futile.

It often feels like a large number of programmers have never really used templates, even if they are C++ programmers

You may be right about that, but many at least – I think most – are using the templates through the STL (or similar foundation libraries) and I suspect that the number of programmers who avoid templates is declining.

Why do you think this is?

Fear of what is different from what they are used to, rumors of code bloat, potential linkage problems, and spectacular bad error messages.

Do you ever wish the GNU C++ compiler provided shorter compiler syntax errors so as to not scare uni students away?

Of course, but it is not all GCC's fault. The fundamental problem is that C++98 provides no way for the programmer to directly and simply state a template's requirements on its argument types. That is a weakness of the language – not of a complier – and can only be completely addressed through a language change, which will be part of C++0x.

I'm referring to *concepts* which will allow C++0x programmers to precisely specify the requirements of sets of template arguments and have those requirements checked at call points and definition points (in isolation) just like any other type check in the language. For details, see any of my papers on C++0x or *Concepts: Linguistic Support for Generic Programming in C++* by Doug Gregor et al (including me) from OOPSLA'06 (available from my publications page). An experimental implementation can be downloaded from Doug Gregor's home pages

(`http://www.osl.iu.edu/~dgregor`).

Until concepts are universally available, we can use *constraint classes* to dramatically improve checking; see my technical FAQ.

The STL is one of the few (if not the only) general purpose libraries for which programmers can actually see complexity guarantees. Why do you think this is?

The STL is – as usual – ahead of its time. It is hard work to provide the right guarantees and most library designers prefer to spend their efforts on more visible features. The complexity guarantees is basically one attempt among many to ensure quality.

In the last couple of years, we have seen distributed computing become more available to the average programmer. How will this affect C++?

That's hard to say, but before dealing with distributed programming, a language has to support concurrency and be able to deal with more than the conventional flat/uniform memory model. C++0x does exactly that. The memory model, the atomic types, and the thread local storage provides the basic guarantees needed to support a good threads library. In all, C++0x allows for the basic and efficient use of multi-cores. On top of that, we need higher-level concurrency models for easy and effective exploitation of concurrency in our applications. Language features such as function objects (available in C++98) and lambdas (a C++0x feature) will help that, but we need to provide support beyond the basic 'let a bunch of threads loose in a common address space' view of concurrency, which I consider necessary as infrastructure and the worst possible way of organizing concurrent applications.

As ever, the C++ approach is to provide efficient primitives and very general (and efficient) abstraction mechanisms, which is then used to build higher-level abstractions as libraries.

Of course you don't have to wait for C++0x to do concurrent programming in C++. People have been doing that for years and most of what the new standard offers related to concurrency is currently available in pre-standard forms.

Do you see this leading to the creation of a new generation of general purpose languages?

Many of the scripting languages provide facilities for managing state in a Web environment, and that is their real strength. Simple text manipulation is fairly easily matched by libraries, such as the new C++ regular expression library (available now from `boost.org`) but it is hard to conceive of a language that is both general-purpose and distributed. The root of that problem is that convenient distributed programming relies on simplification and specialization. A general-purpose language cannot just provide a single high-level distribution model.

I see no fundamental reason against a general-purpose language being augmented by basic facilities for distribution, however, and I (unsuccessfully) argued that C++0x should do exactly that. I think that eventually all major languages will provide some support for distribution through a combination of direct language support, run-time support, or libraries.

Do you feel that resources like the boost libraries will provide this functionality/accessibility for C++?

Some of the boost libraries – especially the networking library – are a good beginning. The C++0x standard threads look a lot like boost threads. If at all possible, a C++ programmer should begin with an existing library (and/or tool), rather than building directly on fundamental language features and/or system threads.

In your opinion, what lasting legacy has C++ brought to computer development?

C++ brought object-oriented programming into the mainstream and it is doing the same for generic programming.

If you look at some of the most successful C++ code, especially as related to general resource management, you tend to find that destructors are central to the design and indispensible. I suspect that the destructor will come to be seen as the most important individual contribution – all else relies on combinations of language features and techniques in the support of a programming style or combinations of programming styles.

Another way of looking at C++'s legacy is that it made abstraction manageable and affordable in application areas where before people needed to program directly in machine terms, such as bits, bytes, words, and addresses.

In the future, I aim for a closer integration of the object-oriented and generic programming styles and a better articulation of the ideals of generality, elegance, and efficiency.

Where do you envisage C++'s future lying?

Much of where C++ has had its most significant strength since day #1: applications with a critical systems programming component, especially the provision of infrastructure. Today, essentially all infrastructures (including the implementation of all higher-level languages) are in C++ (or C) and I expect that to remain the case. Also, embedded systems programming is a major area of use and growth of C++; for example, the software for the next generation US fighter planes are in C++[2]. C++ provides the most where you simultaneously need high performance and higher-level abstractions, especially under resource constraints. Curiously, this description fits both an iPod and a large-scale scientific application.

Has the evolution and popularity of the language surprised you in anyway?

Nobody, with the possible exception of Al Aho (of 'Dragon' book fame), foresaw the scale of C++'s success. I guess that during the 1980s I was simply too busy even to be surprised: The use of C++ doubled every 7.5 months, I later calculated – and that was done without a dedicated marketing department, with hardly any people, and on a shoestring budget. I aimed for generality and efficiency and succeeded beyond anyone's expectations.

By the way, I occasionally encounter people who assume that because I mildly praise C++ and defend it against detractors, I must think it's perfect. That's obviously absurd. C++ has plenty of weaknesses – and I know them better than most – but the whole point of the design and implementation exercise was not to make no mistakes (that's impossible on such a large scale and under such draconian design constraints). The aim was to produce a tool that – in competent hands – would be effective for serious real-world systems building. In that, it succeeded beyond my wildest dreams.

How do you respond to criticism of the language, such as that it has inherited the flaws of C and that it has a very large feature set which makes it bloated?

C++ inherited the weaknesses and the strengths of C, and I think that we have done a decent job at compensating for the weaknesses without compromising the strengths. C is not a simple language (its ISO standard is more than 500 pages) and most modern languages are bigger still. Obviously, C++ (like C) is 'bloated' compared to toy languages, but not really that big compared to other modern languages. There are solid practical reasons why all the languages used for serious industrial work today are 'bloated' – the tasks for which they are used are large and complex beyond the imaginations of ivory tower types.

Another reason for the unpleasantly large size of modern language is the need for stability. I wrote C++ code 20 years ago that still runs today and I'm confident that it will still compile and run 20 years from now. People who build large infrastructure projects need such stability. However, to remain modern and to meet new challenges, a language must grow (either in language features or in foundation libraries), but if you remove anything, you break code. Thus, languages that are built with serious concern for their users (such as C++ and C) tend to accrete features over the decades, tend to become bloated. The alternative is beautiful languages for which you have to rewrite your code every five years.

Finally, C++ deliberately and from day #1 supported more than one programming style and the interaction of those programming styles. If you think that there is one style of programming that is best for all applications and all people – say, object-oriented programming – then you have an opportunity for simplification. However, I firmly believe that the best solutions – the most readable, maintainable, efficient, etc., solutions – to large classes of problems require more than one of the popular programming styles – say, both object-oriented programming and generic programming – so the size of C++ cannot be minimized by supporting just one programming

[2]See the JSF++ coding rules – `http://www.research.att.com/ bs/JSF-AV-rules.pdf` – on my home pages

style. This use of combinations of styles of programming is a key part of my view of C++ and a major part of its strength.

What are you proudest of in terms of the language's initial development and continuing use?

I'm proud that C++ has been used for so many applications that have helped make the world a better place. Through C++, I have made a tiny contribution to the human genome project, to high energy physics (C++ is used at CERN, Fermilab, SLAC, etc.), space exploration, wind energy, etc. You can find a short list of C++ applications on my home pages. I'm always happy when I hear of the language being put to good use.

Secondly, I'm proud that C++ has helped improve the level of quality of code in general – not just in C++. Newer languages, such as JAVA and C#, have been used with techniques that C++ made acceptable for real-world use and compared to code 20 years ago many of the systems we rely on today are unbelievably reliable and have been built with a reasonable degree of economy. Obviously, we can and should do better, but we can take a measure of pride in the progress we have made so far.

In terms of direct personal contribution, I was pleased to be able to write the first C++ compiler, Cfront, to be able to compile real-world programs in 1MB on a 1MHz machine. That is of course unbelievably small by today's standard, but that is what it took to get higher-level programming started on the early PCs. Cfront was written in C with Classes and then transcribed into (early) C++.

Where do you see computer programming languages heading in the near future?

'It is hard to make predictions, especially about the future.' Obviously, I don't really know, but I hope that we'll see general-purpose programming languages with better abstraction mechanisms, better type safety, and better facilities for exploiting concurrency. I expect C++ to be one of those. There will also be bloated corporate infrastructures and languages; there will be special purpose (domain specific) languages galore, and there will be languages as we know them today persisting essentially unchanged in niches. Note that I'm assuming significant evolution of C++ beyond C++0x. I think that the C++ community is far too large and vigorous for the language and its standard library to become essentially static.

Do you have any advice for up-and-coming programmers?

Know the foundations of computer science: algorithms, machine architectures, data structures, etc. Don't just blindly copy techniques from application to application. Know what you are doing, that it works, and why it works. Don't think you know what the industry will be in five years time or what you'll be doing then, so gather a portfolio of general and useful skills. Try to write better, more principled code. Work to make programming more of a professional activity and less of a low-level hacking activity (programming is also a craft, but not just a craft). Learn from the classics in the field and the better advanced textbooks; don't be satisfied with the easily digested how to guides and online documentation – it's shallow.

There's a section of your homepage devoted to 'Did you really say that?' Which quote from this has come back to haunt you the most?

I don't feel haunted. I posted those quotes because people keep asking me about them, so I felt I had better state them clearly. 'C++ makes it harder to shoot yourself in the foot; but when you do, it takes off the whole leg' is sometimes quoted in a manner hostile to C++. That just shows immaturity. Every powerful tool can cause trouble if you misuse it and you have to be more careful with a powerful tool than with a less powerful one: you can do more harm (to yourself or others) with a car than with a bicycle, with a power saw than with a hand saw, etc. What I said in that quote is also true for other modern languages; for example, it is trivial to cause memory exhaustion in a JAVA program. Modern languages are power tools. That's a reason to treat them with respect and for programmers to approach their tasks with a professional attitude. It is not a reason to avoid them, because the low-level alternatives are worse still.

Time for an obligatory question about garbage collection, as we're almost at the end, and you seem to get questions about this all the time. Why do you think

people are so interested in this aspect of the language?

Because resource management is a most important topic, because some people (wrongly) see GC as a sure sign of sloppy and wasteful programming, and because some people (wrongly) see GC as the one feature that distinguishes good languages from inferior ones. My basic view is that GC can be a very useful tool, but that it is neither essential nor appropriate for all programs, so that GC should be something that you can optionally use in C++. C++0x reflects that view.

My view of GC differs from that of many in that I see it as a last resort of resource management, not the first, and that I see it as one tool among many for system design rather than a fundamental tool for simplifying programming.

How do you recommend people handle memory management in C++?

My recommendation is to see memory as just one resource among many (e. g. thread handles, locks, file handles, and sockets) and to represent every resource as an object of some class. For example, memory may be used to hold elements of a container or characters of a string, so we should use types such as `vector<string>` rather than messing around with low-level data structures (e. g. an array of pointers to zero-terminated arrays) and explicit memory management (e. g. `new` and `delete`). Here, both `vector` and `string` can be seen as resource handles that automatically manages the resource that are their elements.

Wherever possible, I recommend the use of such resource handles simply as scoped variables. In that case, there is no explicit memory management that a programmer can get wrong. When an object's lifetime cannot easily be scoped, I recommend some other simple scheme, such as use of smart pointers (appropriate ones provided in C++0x) or representing ownership as membership in some collection (that technique can be used in embedded systems with Draconian time and space requirements). These techniques have the virtues of applying uniformly to all kinds of resources and integrating nicely with a range of error-handling approaches.

Only where such approaches become unmanageable – such as for a system without a definite resource management or error handling architecture or for a system littered with explicit allocation operations – would I apply GC. Unfortunately, such systems are very common, so I consider this is a very strong case for GC even though GC doesn't integrate cleanly with general resource management (don't even think of finalizers). Also, if a collector can be instrumented to report what garbage it finds, it becomes an excellent leak detector.

When you use scoped resource management and containers, comparatively little garbage is generated and GC becomes very fast. Such concerns are behind my claim that 'C++ is my favorite garbage collected language because it generates so little garbage.'

I had hoped that a garbage collector which could be optionally enabled would be part of C++0x, but there were enough technical problems that I have to make do with just a detailed specification of how such a collector integrates with the rest of the language, if provided. As is the case with essentially all C++0x features, an experimental implementation exists.

There are many aspects of garbage collection beyond what I mention here, but after all, this is an interview, not a textbook.

On a less serious note, do you think that facial hair is related to the success of programming languages?

I guess that if we look at it philosophically everything is related somehow, but in this case we have just humor and the fashion of the times. An earlier generation of designers of successful languages was beardless: Backus (Fortran), Hopper (COBOL), and McCarthy (Lisp), as were Dahl and Nygaard (Simula and object-oriented programming). In my case, I'm just pragmatic: while I was living in colder climates (Denmark, England, and New Jersey), I wore a beard; now I live in a very hot place, Texas, and choose not to suffer under a beard. Interestingly, the photo they use to illustrate an intermediate stage of my beard does no such thing. It shows me visiting Norway and reverting to cold-weather type for a few days. Maybe there are other interesting correlations? Maybe there is one between designer height and language success? Maybe there is a collation between language success and appreciation of Monty Python? Someone could have fun doing a bit of research on this.

Finally, is there anything else you'd like to add?

Yes, I think we ought to consider the articulation of ideas and education. I have touched upon those topics a couple of times above, but the problems of getting people to understand what C++ was supposed to be and how to use it well were at least as difficult and time consuming as designing and implementing it. It is pointless to do good technical work and then not tell people about it. By themselves, language features are sterile and boring; to be useful, programmers have to learn how language features can be used in combination to serve some ideal of programming, such as object-oriented programming and generic programming.

I have of course written many purely technical papers, but much of my writing have been aimed at raising the abstraction level of programs, to improve the quality of code, and to give people an understanding of what works and why. Asking programmers to do something without giving a reason is treating them like small children – they ought to be offended by that. The editions of *The C++ Programming Language*, *D&E*, *Teaching Standard C++ as a New Language*, and my HOPL papers are among my attempts to articulate my ideals for C++ and to help the C++ community mature. Of course, that has been only partially successful – there is still much cut-and-paste programming being done and no shortage of poor C++ code – but I am encouraged by the amount of good code and the number of quality systems produced.

Lately, I have moved from industry to academia and now see the education problems from a different angle. We need to improve the education of our software developers. Over the last three years, I have developed a new course for freshmen (first-year students, often first-time programmers). This has given me the opportunity to address an audience I have never before known well and the result is a beginner's textbook *Programming: Principles and Practice using C++* which will be available in October.

Clojure: Rich Hickey

Clojure's creator, Rick Hickey, took some time to tell Computerworld about his choice to create another Lisp dialect, the challenges of getting Clojure to better compete with Java and C#, and his desire to see Clojure become a 'go-to' language

What prompted the creation of Clojure?

After almost 20 years of programming in C++/Java/C#, I was tired of it. I had seen how powerful, dynamic and expressive Common Lisp was and wanted to have that same power in my commercial development work, which targeted the JVM/CLR. I had made a few attempts at bridging Lisp and Java, but none were satisfying. I needed something that could deploy in a standard way, on the standard platforms, with very tight integration with existing investments.

At the same time, throughout my career I have been doing multithreaded programming, things like broadcast automation systems, in these OO languages, and seen nothing but pain. As a self-defense and sanity-preserving measure, I had moved my Java and C# code to a non-OO, functional style, emphasising immutability. I found this worked quite well, if awkward and non-idiomatic.

So, I wanted a dynamic, expressive, functional language, native on the JVM/CLR, and found none.

Where does the name Clojure come from?

It's a pun on the closure programming construct (and is pronounced identically). I wanted a name that involved C (CLR), L (Lisp) and J (JVM). There were no search hits and the domain was available – what's not to like?

Was there a particular problem the language aimed to solve?

Clojure is designed to support writing robust programs that are simple and fast. We suffer from so much incidental complexity in traditional OO languages, both syntactic and semantic, that I don't think we even realise it anymore. I wanted to make 'doing the right thing' not a matter of convention and discipline, but the default. I wanted a solid concurrency story and great interoperability with existing Java libraries.

Why did you choose to create another Lisp dialect instead of extending an existing one?

While Lisps are traditionally extremely extensible, I had made some design decisions, like immutability for the core data structures, that would have broken backward compatibility with existing Scheme and Common Lisp programs. Starting with a clean slate let me do many other things differently, which is important, since I didn't want Clojure to appeal only to existing Lispers. In the end Clojure is very different and more approachable to those having no Lisp background.

Why did you pick the JVM?

I originally targeted both the JVM and CLR, but eventually decided I wanted to do twice as much, rather than everything twice. I chose the JVM because of the much larger open source ecosystem surrounding it and it has proved to be a good choice. That said, the CLR port has been revitalised by David Miller, is an official part of the Clojure project and is approaching feature-parity with the JVM version.

Clojure-in-Clojure: self-hosting is usually a big milestone for programming languages – how is that going?

It is going well. We are approaching the end of the foundation-laying phase. There were a few base capabilities of Java which I leveraged in the implementation of Clojure for which there was no analogy in Clojure itself. Now the last of these is coming into place. Then there will be nothing precluding the implementation of the Clojure compiler and the Clojure data structures in Clojure itself, with efficiency equivalent to the original Java implementation.

Did you run into any big problems while developing the language?

One of the biggest challenges was getting the persistent data structures right, with sufficient performance such that Clojure could be a viable alternative to Java and C#. Without that, I wouldn't have gone forward.

We've all read *The rise of 'Worse is Better'* by Richard Gabriel. Do you feel that a project like Clojure can help reverse that attitude?

The arguments made in *Worse is Better* are very nuanced and I'm not sure I understand them all, so Clojure tries to take both sides! It values simplicity of interface and of implementation. When there is a conflict, Clojure errs on the side of pragmatism. It is a tool, after all.

With multi-core CPUs becoming more common and a resurgence of hyperthreading, dealing with concurrent tasks is now more important. How does Clojure deal with this?

Good support for concurrency is a central feature of Clojure. It starts with an emphasis on functional programming. All of the core data structures in Clojure are immutable, so right off the bat you are always working with data that can be freely shared between threads with no locking or other complexity whatsoever, and the core library functions are free of side-effects. But Clojure also recognises the need to manage values that differ over time. It supports that by placing values in references, which both call out their stateful nature and provide explicit concurrency semantics that are managed by the language.

For example, one set of references in Clojure are transactional, which lets you conduct database-like transactions with your in-memory data and, like a database, automatically ensures atomic/consistent/isolated integrity when multiple threads contend for the same data. In all cases, Clojure's reference types avoid the complications and deadlocks of manual locking.

What can you tell us about the support for parallelism and the upcoming Java ForkJoin framework?

While concurrency focuses on coordinating multiple tasks, parallelism focuses on dividing up a single task to leverage these multi-cores to get the result faster. I didn't build any low-level infrastructure for parallelism into Clojure since the Java concurrency experts were already doing that in the form of the ForkJoin framework, a sophisticated thread pool and work-stealing system for parallel computation. As that framework is stabilising and moving towards inclusion in Java 7 (and usable with Java 6), I've started implementing parallel algorithms, like mapping a function across a vector by breaking it into subtasks, using ForkJoin. Clojure's data structures are well suited for this decomposition, so I expect to see a rich set of parallel functions on the existing data structures – i. e., you won't have to use special 'parallel' data structures.

What about running on distributed systems? MapReduce did come from Lisp ...

I don't think distribution should be hardwired into a general purpose programming language. Clojure can tap into the many options for distribution on the JVM – JMS, Hadoop, Terracotta, AMQP, XMPP, JXTA, JINI, JGroups etc, and people are already leveraging many of those.

How did you choose the Eclipse License for Clojure?

The EPL has the advantage of being reciprocal without impinging on non-derivative work with which it is combined. Thus, it is widely considered to be commercial-friendly and acceptable for businesses.

Web frameworks? I notice there's one called 'Compojure.' Do you see this as a direction in which Clojure could grow?

Definitely, there are already interesting frameworks for Clojure in many areas. One of the nice things about libraries for Clojure is that they can leverage tried-and-true Java libraries for the low-level plumbing and focus on higher-level use and flexibility.

What books would you recommend for those wanting to learn Clojure?

Programming Clojure, by Stuart Halloway, published by Pragmatic Programmers is the book right now and it's quite good – concise and inspiring, I highly recommend it. I know of a couple of other books in the works.

What's the most interesting program(s) you've seen written with Clojure?

There are a bunch of start-ups doing interesting things I'm not sure I can talk about. Clojure has been applied so diversely, given its youth – legal document processing, an R-like statistical language, and a message routing system in a veterinary hospital, for example.

You recently released Clojure 1.0. What features were you the most excited about?

Version 1.0 was less about new features than it was about stability. For example, the feature base was sufficient that people weren't lacking anything major for doing production work and it could serve as a baseline for Stuart's book.

Has hosting the project on GitHub helped you increase the number of contributors and the community around Clojure?

The contributor list has been growing steadily. I think being on GitHub makes it easier for people to work on contributions.

I understand you started working on Clojure during a sabbatical. How has the situation changed now?

I'd like to continue to work on Clojure full-time but in order to do so I need to find an income model. I can't say I've figured that out yet but, as Clojure gets more widespread commercial adoption, I'm hoping for more opportunities.

Perl gurus are 'Perl Mongers,' Python ones are 'Pythonistas.' We think Clojure needs something similar. Any suggestions?

I think everyone has settled on Clojurians.

What is it with Lisp programmers and nested lists?

Programming with data structures might be unfamiliar to some but it is neither confusing nor complex once you get over the familiarity hump. It is an important and valuable feature that can be difficult to appreciate until you've given it a try.

This question must be asked ... What's the highest number of closing brackets you've seen in a row?!

What brackets?! I don't see them anymore and neither do most Clojure developers after a short time. One advantage of piling them up is that the code ends up being denser vertically so you can see more of the logic in one screen, versus many lines of closing }'s (Java et al) or `end`'s (Ruby).

Looking back, is there anything you would change in the language's development?

I think it's quite important that a significant portion of Clojure's design was informed by use, and continues to be so. I'm happy with the process and the outcome.

Where do you envisage Clojure's future lying?

Right now we're in the early adopter phase, with startups and ISVs using Clojure as a secret weapon and power tool. Clojure is a general purpose language and already being applied in a wide variety of domains. It's impossible to predict but I'd be very happy to see Clojure become a go-to language when you want the speed of dynamic development coupled with robustness, performance and platform compatibility.

What do you think will be Clojure's lasting legacy?

I have no idea. It would be nice if Clojure played a role in popularising a functional style of programming.

D: Walter Bright

According to his home page, Walter Bright was trained as a mechanical engineer, and has worked for Boeing on the 757 stabilizer trim system. Ever since this experience however, he has been writing software, and has a particular interest in compilers. We chat to Walter about D and his desire to improve on systems programming languages

What prompted the development of D?

Being a compiler developer, there's always at the back of my mind the impetus for applying what I know to design a better language. At my core I'm an engineer, and never can look at anything without thinking of ways to improve it.

The tipping point came in 1999 when I left Symantec and found myself at a crossroad. It was the perfect opportunity to put into practice what I'd been thinking about for many years.

Was there a particular problem you were trying to solve?

There was no specific problem. I'd been writing code in C++ for 12 years, and had written a successful C++ compiler. This gave me a fairly intimate knowledge of how the language worked and where the problems were.

C++ was (and is) limited by the requirement of legacy compatibility, and I thought much could be done if that requirement was set aside. We could have the power of C++ with the hindsight to make it beautiful.

I had also been programming in other languages, which had a lot to contribute.

How did the name D come about?

It started out as the Mars programming language (as the company name is Digital Mars). But my friends and colleagues kept calling it D, as it started out as a re-engineering of C++, and eventually the name stuck.

Why did you feel that C++ needed re-engineering?

A lot has been learned about programming since C++ was developed. Much of this has been folded in C++ as layers on top of the existing structure, to maintain backwards compatibility. It's like a farmhouse that has been added on to by successive generations, each trying to modernize it and adapting it to their particular needs. At some point, with a redesign, you can achieve what is needed directly.

But D today has moved well beyond that. Many successful concepts from other languages like JavaScript, Perl, Ruby, Lisp, Ada, Erlang, Python, etc., have had a significant influence on D.

What elements of C++ have you kept, and what elements have you deliberately discarded?

D keeps enough so that a C++ programmer would feel immediately comfortable programming in D. Obsolete technologies like the preprocessor have been replaced with modern systems, such as modules. Probably the central thing that has been kept is the idea that D is a systems programming language, and the language always respects that ultimately the programmer knows best.

Would you do anything differently in the development of D if you had the chance?

I'd be much quicker to turn more of the responsibility for the work over to the community. Letting go of things is hard for me and something I need to do a lot better job of.

What is the most interesting program that you've seen written with D?

Don Clugston wrote a fascinating program that was actually able to generate floating point code and then execute it. He discusses it in this presentation:
`http://video.google.com/videoplay?docid=-1440222849043528221&hl=en`.

What sort of feedback have you received from the experimental version of D, or D 2.0, released in June 2007?

D 1.0 was pretty straightforward stuff, being features that were adapted from well-trod experience in other languages. D 2.0 has ventured into unexplored territory that doesn't have a track record in other languages. Since these capabilities are unproven, they generate some healthy scepticism. Only time will tell.

Have you ever seen the language used in a way that was not originally intended? If so, what was it? And did it or didn't it work?

There have been too many to list them all, but a couple examples are Don Clugston and Andrei Alexandrescu. They never cease to amaze me with how they use D. They often push the language beyond its limits, which turns into some powerful motivation to extend those limits and make it work. Don's presentation in the afore-mentioned video is a good example. You can see a glimpse of Andrei's work in, for example, the algorithms library at
`http://www.digitalmars.com/d/2.0/phobos/std_algorithm.html`.

Do you still consider D to be a language under development?

A language that is not under development is a language that is not being used. D is under development, and will stay that way as long as people use it. C++, Java, Python, Perl, etc., are also widely used and are still under development.

Are changes still being made to the language or are you focusing on removing bugs right now?

I spend about half of my efforts fixing bugs and supporting existing releases, and the other half working on the future design of D 2.0.

Do you agree that the lack of support from many IDEs currently is a problem for the language's popularity right now?

There are many editors and IDEs that support D now:
`http://www.prowiki.org/wiki4d/wiki.cgi?EditorSupport`.

How do you react to criticism such as the comment below, taken from Wikipedia: '*The standard library in D is called Phobos. Some members of the D community think Phobos is too simplistic and that it has numerous quirks and other issues, and a replacement of the library called Tango was written. However, Tango and Phobos are at the moment incompatible due to different run-time libraries (the garbage collector, threading support, etc). The existence of two libraries, both widely in use, could lead to significant problems where some packages use Phobos and others use Tango.*' ?

It's a valid criticism. We're working with the Tango team to erase the compatibility issues between them, so that a user can mix and match what they need from both libraries.

In your opinion, what lasting legacy has D brought to computer development?

D demonstrates that it is possible to build a powerful programming language that is both easy to use and generates fast code. I expect we'll see a lot of D's pioneering features and feature combinations appearing in other languages.

Where do you envisage D's future lying?

D will be the first choice of languages for systems and applications work that require high performance along with high programmer productivity.

Where do you see computer programming languages heading in the future, particularly in the next 5 to 20 years?

The revolution coming is large numbers of cores available in the CPUs. That means programming languages will have to adapt to make it much easier to take advantage of those cores. Andrei's presentation `http://www.digitalmars.com/d/2.0/accu-functional.pdf` gives a taste of what's to come.

Do you have any advice for up-and-coming programmers?

Ignore all the people who tell you it can't be done. Telling you it can't be done means you're on the right track.

Is there anything else you'd like to add?

Yes. D isn't just my effort. An incredible community has grown up around it and contribute daily to it. Three books are out on D now and more are on the way. The community has created and released powerful libraries, debuggers, and IDEs for D. Another D compiler has been created to work with gcc, called gdc, and a third is being developed for use with LLVM. Proposals for new language features appear almost daily. D has an embarrassment of riches in the people contributing to it.

Oh, and I'm having a great time.

Erlang: Joe Armstrong

Erlang creator Joe Armstrong took some time to tell Computerworld about Erlang's development over the past 20 years, and what's in store for the language in the future

What's behind the name Erlang?

Either it's short for 'Ericsson Language' or it's named after the Danish mathematician Agner Krarup Erlang. We have never revealed which of these is true, so you'll have to keep guessing!

What prompted its creation?

It was an accident. There was never a project 'to create a new programming language.' There was an Ericsson research project 'to find better ways of programming telephony applications' and Erlang was the result.

Was there a particular problem the language aimed to solve?

Yes, we wanted to write a control program for a small telephone exchange in the best possible manner. A lot of the properties of Erlang can be traced back to this problem. Telephone exchanges should never stop, so we have to be able to upgrade code without stopping the system.

The application should never fail disastrously so we needed to develop sophisticated strategies for dealing with software and hardware errors during run-time.

Why was Erlang released as open source? What's the current version of open source Erlang?

To stimulate the spread of Erlang outside Ericsson. The current version is release 13 – so it's pretty mature. We release about two new versions per year.

What's the Erlang eco-system like?

There's a very active mailing list where we have a lot of discussions about architectures and applications and help solve beginners problems.

Currently there are several conferences which are dedicated to Erlang. The oldest is the *Erlang User Conference* that runs once a year in Stockholm. The *ACM Functional Programming Conference* has had an 'Erlang day' for the last few years and last year the 'Erlang Factory' started.

The *Erlang Factory* runs twice a year. The last conference was in Palo Alto and the next one will be in London. These conferences are explosions of enthusiasm. They are to become the meeting place for people who want to build large scale systems that never stop.

It's difficult to get overall picture. Erlang is best suited for writing fault-tolerant servers. These are things that are not particularly visible to the end-user. If you have a desktop application, it's pretty easy to find out how it's been implemented. But for a server this is much more difficult. The only way to talk to a server is through an agreed protocol, so you have no idea how the server has been implemented.

What's the most interesting program(s) you've seen written with Erlang for business?

That's difficult to answer, there are many good applications.

Possibly Ejabberd which is an open-source Jabber/XMPP instant messaging server. Ejabberd appears to be the market leading XMPP server and things like Google Wave which runs on top of XMPP will probably attract a lot of people into building applications on XMPP servers.

Another candidate might be Rabbit MQ which is an open-source implementation of the AMQP protocol. This provides reliable persistent messaging in a language-neutral manner. Building systems without shared memory and based on pure message passing is really the only way to make scalable and reliable systems. So AMQP fits nicely with the Erlang view of the world.

How flexible is the language, how does it stand aside the popular programming languages for general applications?

Difficult to say. What we lose in sequential performance we win back in parallel performance. To fully utilize a multicore or cloud infrastructure your program must exploit parallelism. A sequential program just won't run faster on a multicore, in fact as time goes on it will run slower since clock speeds will drop in the future to save power. The trend is towards more and slower cores. The ease of writing parallel programs is thus essential to performance.

In the ERLANG world we have over twenty years of experience with designing and implementing parallel algorithms. What we lose in sequential processing speed we win back in parallel performance and fault-tolerance.

Have you ever seen the language used in a way that wasn't originally intended?

Lots of times ...

What limits does Erlang have?

You have to distinguish the language from the implementation here. The implementation has various limits, like there is an upper limit on the maximum number of processes you can create; this is very large but is still a limit.

Somewhere in the next 10 to 20 years time we might have a million cores per chip and Petabyte memories and will discover that 'hey – we can't address all this stuff' so we'll have to change the implementation – but the language will be the same.

We might discover that massive programs running 'in the cloud' will need new as yet unthought of mechanism, so we might need to change the language.

Were there any particularly difficult or frustrating problems you had to overcome in the development of the language?

Yes. An engineer's job is to solve problems. That's why I'm an engineer. If the problems weren't difficult they would be no point in doing the job [but] 95 percent of the time the problems are in a state of 'not being solved' which is frustrating. Frustration goes hand-in-hand with creativity – if you weren't frustrated with how things worked you would see no need to invent new things.

What is the third-party library availability like?

Patchy. In some areas it's absolutely brilliant, in others non-existent. This is a chicken and egg situation. Without a lot of active developers there won't be many third-party libraries, and without a lot of libraries we won't attract the developers.

What's happening is that a lot of early-adopters are learning ERLANG and using it for things that we hadn't imagined. So we're seeing things like CouchDB (a database) and MochiWeb (a Web server) which you can use to build applications.

Programming languages are leveraging more and more threading due to multicore processors. Will this push the development of Erlang?

Very much so. We've been doing parallel programming since 1986 and now we have real parallel hardware to run our programs on, so our theories are turning into reality. We know how to write parallel programs, we know how to deploy them on multicores. We know how to debug our parallel programs. We have a head start here.

What we don't know is the best way to get optimal performance from a multicore so we're doing a lot of tweaking behind the scenes.

The good news for the ERLANG programmer is that they can more or less ignore most of the problems of multicore programming. They just write ERLANG code and the ERLANG run-time system will try and spread the execution over the available cores in an optimal manner.

As each new version of ERLANG is released we hope to improve the mapping onto multicores. This is all highly dynamic, we don't know what multicore architectures will win in the future. Are we going to see small numbers of complex cores or large numbers of simple cores with a 'network on chip' architecture (as in the Tilera chips, or the Intel Polaris chip)? We just don't know.

But whatever happens ERLANG will be there adapting to the latest chipsets.

Did you see this trend coming in the early days of its development?

No. We always said 'one day everything will be parallel' – but the multi-core stuff sneaked up

when we weren't watching. I guess the hardware guys knew about this in advance but the speed with which the change came was a bit of a surprise. Suddenly my laptop had a dual-core and a quad-core appeared on my desktop.

And wow – when the dual core came some of my ERLANG programs just went twice as fast with no changes. Other programs didn't go twice as fast. So the reasons why the program didn't go twice as fast suddenly became a really interesting problem.

What are the advantages of hot swapping?

You're joking. In my world we want to build systems that are started once and thereafter never stop. They evolve with time. Stopping a system to upgrade the code is an admission of failure.

ERLANG takes care of a lot of the nitty-gritty details needed to hot-swap code in an application. It doesn't entirely solve the problem, since you have to be pretty careful if you change code as you run it, but the in-build mechanisms in ERLANG make this a tractable problem.

Functional versus imperative? What can you tell us?

It's the next step in programming. Functional programs to a large extent behave like the maths we learned in school.

Functional programming is good in the sense that it eliminates whole classes of errors that can occur in imperative programs. In pure functional programs there is no mutable data and side effects are prohibited. You don't need locks to lock the data while it is being mutated, since there is no mutation. This enables concurrency, all the arguments to any function can be evaluated in parallel if needed.

Interpreted versus compiled? Why those options?

I think the distinction is artificial. ERLANG is compiled to abstract machine code, which is then interpreted. The abstract machine code can be native code compiled if necessary. This is just the same philosophy as used in the JVM and .NET.

Whether or not to interpret or compile the code is a purely engineering question. It depends upon the performance, memory-size, portability etc. requirements we have. As far as the user is concerned their is no difference. Sometimes compiled code is faster than interpreted code, other times it is slower.

Looking back, is there anything you would change in the language's development?

Removing stuff turns out to be painfully difficult. It's really easy to add features to a language, but almost impossibly difficult to remove things. In the early days we would happily add things to the language and remove them if they were a bad idea. Now removing things is almost impossible.

The main problem here is testing, we have systems with literally millions of lines of code and testing them takes a long time, so we can only make backwards compatible changes.

Some things we added to the language were with hindsight not so brilliant. I'd happily remove macros, include files, and the way we handle records. I'd also add mechanism to allow the language itself to evolve.

We have mechanisms that allow the application software to evolve, but not the language and libraries itself. We need mechanisms for revision control as part of the language itself. But I don't know how to do this. I've been thinking about this for a long time.

Instead of having external revision control systems like Git or Subversion I'd like to see revision control and re-factoring built into the language itself with fine-grain mechanism for introspection and version control.

Will computer science students finally have to learn about dining philosophers?!

Easy – give 'em more forks.

Finally, where do you envisage Erlang's future lying?

I don't know. ERLANG destiny seems to be to influence the design of future programming languages. Several new programming languages have adopted the ERLANG way of thinking about concurrency, but they haven't followed up on fault-tolerance and dynamic code-change mechanisms.

As we move into cloud computing and massively multicores life becomes interesting. How do we program large assemblies of parallel processes? Nobody really knows. Exactly what is a cloud? Again nobody knows.

I think as systems evolve ERLANG will be there someplace as we figure out how to program massively fault-tolerant systems.

F#: Don Syme

Microsoft researcher Don Syme talks about the development of F#, its simplicity when solving complex tasks, the thriving F# community and the future ahead for this fuctional programming language

What prompted the development of F#?

From the beginning, the aim of F# has been to ensure that typed functional programming in the spirit of OCaml and Haskell, finds a high-quality expression on the .NET framework. These languages excel in tasks such as data transformations and parallel programming, as well as general purpose programming.

How did the name F# come about?

In the F# team we say 'F is for Fun.' Programming with F# really does make many routine programming tasks simpler and more enjoyable, and our users have consistently reported that they've found using the language enjoyable.

However, in truth the name comes from 'F for Functional,' as well as a theoretical system called 'System F.'

Were there any particular problems you had to overcome in the development of the language?

Combining object-oriented and functional programming poses several challenges, from surface syntax to type inference to design techniques. I'm very proud of how we've addressed these problems.

F# also has a feature called 'computation expressions,' and we're particularly happy with the unity we've achieved here.

Would you have done anything differently in the development of F# if you had the chance?

In a sense, we're tackling this now. Some experimental features have been removed as we're bringing F# up to product quality, and we've also made important cleanups to the language and library. These changes have been very welcomed by the F# community.

Was F# originally designed in the .NET framework?

Yes, totally. F# is all about leveraging the benefits of both typed functional programming and .NET in unison.

What elements has F# borrowed from ML and OCaml?

F# is heavily rooted in OCaml, and shares a core language that permits many programs to be cross-compiled. The type system and surface syntax are thus heavily influenced by OCaml.

What feedback did the F# September 2008 CTP release get?

It's been really great. We've heard from existing F# developers who have been really happy to see all the improvements in the CTP release – in particular some of the improvements in the Visual Studio integration. It's also been great to see lots of new users coming to F# with this new release.

Do you have any idea how large the F# community currently is?

It's hard to tell. We're getting an excellent and active community developing, mainly around *hubFS* and have seen consistent growth throughout the year.

You say on your blog that 'one of the key things about F# is that it spans the spectrum from interactive, explorative scripting to component and large-scale software development.' Was this always a key part of the development of F#, or has it simply morphed into a language with these features over time?

A key development for us was when we combined F# Interactive with Visual Studio. This allowed F# users to develop fast, accurate code using Visual Studio's background type-checking

and Intellisense, while interactively exploring a problem space using F# Interactive. We brought these tools together in late 2005, and that's when the language really started hitting its niche.

What are you currently most excited about in the development of F#?

This year we have really focused on ensuring that programming in F# is simple and intuitive. For example, I greatly enjoyed working with a high-school student who learned F#. After a few days she was accurately modifying a solar system simulator, despite the fact she'd never programmed before. You really learn a lot by watching a student at that stage.

How much influence has Haskell had on the development of F#?

A lot! One of the key designers of Haskell, Simon Peyton-Jones, is just down the corridor from me at Microsoft Research Cambridge and has been a great help with F#, so I have a lot to thank him for. Simon gave a lot of feedback on the feature called 'asynchronous workflows' in particular. The F# lightweight syntax was also inspired by Haskell and Python. Over the last five years F# has seen a lot of idea sharing in the language community, at conferences such as Lang.NET. The .NET framework has played an important role in bringing the programming camps together.

Have you always worked with functional languages? Do you have a particular affinity with them? What initially attracted you?

I've worked with many languages, from BASIC to assembly code. One of the last check-ins I made when implementing generics for .NET, C# and VB had a lot of x86 assembly code. My first job was in Prolog. I think programmers should learn languages at all extremes.

Functional languages attract me because of their simplicity even when solving complex tasks. If you look through the code samples in a book such as *F# for Scientists* they are breathtaking in their elegance, given what they achieve. A good functional program is like a beautiful poem: you see the pieces of a 'solution' come together.

Of course, not all programs end up so beautiful. It's very important that we tackle 'programming in the large' as well. That's what the object-oriented features of F# are for.

Why did Microsoft decide to undertake the development of F# and how does F# fit into Microsoft's overall strategy and philosophy?

Microsoft's decision to invest in further F# research is very much based on the fact that F# adds huge value to the .NET platform. F# really enables the .NET platform to reach out to new classes of developers, and appeal to domains where .NET is not heavily used. This is especially true in data exploration and technical computing. We're also exploiting functional techniques in parallel programming.

What is the most interesting program you've seen written in F#?

That's a good question! I'll give several answers. I've mentioned the samples from *F# for Scientists*, which are very compelling. But for sheer F# elegance, I like Dustin Campbell's series of Project Euler solutions.

However, some of the most intriguing to me are the ones that integrate F# into existing data-oriented tools such as AutoCAD and ArcGIS. These domains are, in theory, well suited to functional programming, but no functional language has ever interoperated with these tools before. Through the magic of .NET interoperability, you can now use F# with the .NET APIs for these tools, which opens up many possibilities.

Why do you think a programmer would choose to write apps in F# rather than C#?

Many programmers choose to explore a problem in F# because it lets them focus more on the problem domain and less on programming itself. That's a big benefit in some of the data exploration, algorithmic and technical computing domains, and so we've seen a lot of interest in using F# here, where C# may not have been an obvious choice.

Do you think that F# and C# are complimentary languages, or will one become more dominant than the other?

C# and VB.NET are clearly the elder statesmen of.NET languages and it's hard to imagine a

really major .NET project where these languages don't play a significant role. So the approach we take with F# is that it's definitely complementary to C#. We expect there will be many F# projects that incorporate C# components. For example, the designer tools we use with F# emit C# code, and you then call your F# code from those the event handlers. A working knowledge of C# is thus very useful for the F# programmer.

In your opinion, what lasting legacy will F# bring to computer development?

Our aim with F# has been to make typed functional programming real and viable. The feedback we've received often shows that our users are thrilled to have a programming solution that fills this role. However, perhaps the greatest sign of success will be when people copy what we've done and reuse the ideas in other settings.

Have you received much criticism of the language so far? If so, what has this been focused on?

We've received lots and lots of feedback – we've been in almost continual engagement with the F# community for the last three years. This has been extraordinary. People have been very helpful, and have come up with many great ideas and suggestions. However, we're just as glad to get the 'this is broken' emails as we are glowing praise – indeed even more glad – we want to know when things don't add up, or don't make sense.

Some programmers do have a hard time adjusting their mindset from imperative programming to OO, though most find the transition enjoyable. Learning new paradigms can sometimes be easier for beginners than experienced programmers. However, one of the great things about F# is that you can 'change one variable at a time,' e. g. continue to use your OO design patterns, but use functional programming to implement portions of your code.

What are you proudest of in terms of the language's initial development and continuing use?

I'm proud of the F# community, for being so helpful to beginners. For the language itself, I'm very happy with the way we've stayed true to functional programming while still integrating with .NET.

Where do you see computer programming languages heading in the future?

People thirst for simplicity. People need simple solutions to the problems that really matter: data access, code development, deployment, cloud computing, Web programming, and parallel programming, to name a few. One of the exciting things about working in the Visual Studio team is that there are world experts in all of these domains working in unison. We won't cover all of these bases with the first release of F#, but over time we'll be operating in all these domains.

At the language level, people say that languages are converging in the direction of mixed functional/OO programming. However, I expect this will enable many important developments on the base. For example, I'm a proponent of language-integrated techniques that make it easier to express parallel and asynchronous architectures.

Do you have any advice for up-and-coming programmers?

Learn F#, Python, Prolog, Haskell, C# and Erlang!

Falcon: Giancarlo Niccolai

Falcon's creator Giancarlo Niccolai took some time to tell Computerworld about the development of Falcon, the power and influence of C++, and how the new multithreading design in Falcon version 0.9 will innovate the scripting language panorama

What prompted the creation of Falcon?

Part of my daily job was taking care of the status of servers streaming real time data through financial networks. A scripting facility in the control application would have been a godsend, as the alert conditions were too complex to be determined, constantly shifting and requiring constant supervision. I was not new to the topic, as I previously worked on the xHarbour project (a modern porting of the xBase languages), and I also did some research in the field.

The workload was heavy; even if the logic to be applied on each message was simple, data passing through was in the order of thousands of messages per second, each requiring prompt action, and each composed of about one to four kilobytes of raw data already de-serialised into complex C++ class hierarchies.

In terms of raw calculation power, the existing engines were adequate, but they were greedy. They considered their task as the most important thing to carry on the whole application, and so they didn't care very much about the time needed to setup a script, to launch a callback, to wrap external data or to provide them with data coming from the application at a very high rate.

It was also quite hard to use them in a multithread context. The only VM designed to work in multithreading (that is, to be used concurrently by different threads – we had many connections and data streams to take care of) was Python, but it worked with a very primitive concept of multithreading forcing global locks at every allocation and during various steps of each script. My test showed that this caused rapid slow down, even in parts of the application not directly related with the script (i. e., in the parts preparing the data for the scripts before launching Python VMs).

Of all the possible scripting engines, Lua was the most adequate, but using it from concurrent threads posed some problems (at the time, the memory allocator wasn't threadsafe, and needed to be refitted with wide global locks). Also, having to deal with wide integer data (prices on financial markets are often distributed as int64 with decimal divisor) I was worried about the fact that Lua provided only one type of numbers – 58-bit precision floating point. These also caused severe rounding/precision problems in the financial area, and are thus generally avoided.

There was so much work to do on those engines to make them able to meet the requirements for task that the alternatives were either dropping the idea of scripting the control application, and then the servers themselves on a second stage, or writing something new.

I hate to give up, so I started to work at HASTE (Haste Advanced Simple Text Evaluator), a scripting engine meant to be just a scripting engine, and to drive massive throughput of data with the host application.

Was there a particular problem the language aimed to solve?

The main idea behind the early development of HASTE was the 'integratability' with existing complex multithreaded applications and the interaction with real-time, urgent and massive data flow.

When I had something working, I soon realised that the ability to deal with raw data and the way the VM cooperated with the host application was very precious in areas where other scripting languages didn't shine like binary file parsing, image manipulation, gaming (not game scripting), wide/international string manipulation, etc.

At the same time, I found the constructs in other scripting languages limiting. I could live with them, as imperative and prototype-based programming are quite powerful, and the 'total OOP' approach of Ruby is fascinating, but now that I had HASTE working and doing fine (the HASTE VM was simpler and slightly faster than Lua's) I started thinking beyond the pure

needs of the engine.

As a professional, that exact task for which I built HASTE was just a small part of my daily activities. Similarly to the way Larry Wall built Perl out of his needs (to parse a massive amount of unstructured log dataset), I started to feel the need for higher logic to carry on my tasks – complex analysis on structured data, pattern finding and decision making.

I used to work with many languages including C, C++, Java, assembly, Lisp, Prolog, Clipper/xBase, Delphi, SQL, and of course Python, Lua, Perl and PHP, and I learned through time to employ the best instruments to solve the problem at hand. I felt the need for a tool flexible enough that it could cover my daily needs and drive new ideas.

Pure ideas are useful for the machine. A pure logic language, such as Prolog, can explode the rules into raw machine code, being as fast as possible in finding solutions. Pure functional languages, such as Erlang, can parallelize massive calculation automatically at compile time. Pure OOP languages, such as Ruby, can treat any entity just the same, reducing the complexity of the code needed to implement them.

But purity is never a good idea for the mind. The mind works towards unification and analogy, and solutions in the real world are usually more effective when a wide set of resolutive techniques can be employed. This is true even in mathematics, where you need to apply different resolutive techniques (and often also a good deal of fantasy and experience) to solve seemingly 'mechanical' problems as the reduction of a differential equation. HASTE was terribly simple, a purely procedural language with arrays and dictionaries, but it had an interesting feature – functions were considered normal items themselves. This gave me the idea of working towards a general purpose language (beyond the scripting engine of HASTE) whose 'programming paradigm' was 'all and none.'

So Falcon was born with the idea of having pure OOP (so that raw, hard C structures could be mapped into it without the need for a dictionary-like structure to be filled), but not being OOP, with the idea of having pure procedural structure (driven by old, dear functions and call-return workflow) but not being procedural, and with the idea of having functional constructs, but without being functional.

It was also developed with the idea to add new ideas into it besides, and throughout, the existing ideas. A set of expanding concepts to solve problems with new conceptual tools, to reduce the strain needed by the professional developer to find the right way to match its idea with the problem and to create a living solution. Not needing anymore to learn how to think in a language to have it to work out the solution, but having a language moving towards the way the programmer's mind solves problems. I needed a tool through which I could shape easily and flexibly solutions to higher logic problems, in different and ever shifting domains (one day parse gigabytes of binary data in search for behaviour patterns, the other day organising classroom 'turnations' for courses in my company) and to do that fast.

If there is one thing I'm proud of in Falcon it's that it wasn't born for the most exotic reasons, but to address the problem of integration and empowerment of massive applications on one side and the necessity do solve complex logic and highly mutable problems on the other. Or in other words, it was born as a necessary tool.

Few languages were born to address real problems, like Perl, Clipper, possibly C++ and C which actually evolved from research/didactic university projects.

Why did you choose C++ to base Falcon on, rather than a lower-level language? What are the similarities between the two languages?

I like OOP and I like the power of C. When I decided to go C++ for Falcon, I was positive that I would have used C for the low-level stuff and C++ classes to shape higher concepts. All the applications I had to script were C++, or could integrate with C++. Also, I was told by a friend of mine working in the gaming industry that virtual calls were actually more efficient than switches, and we have lots of them in a scripting language. I tested the thing out for myself, and it turned out that modern processors are performing really well in presence of virtual calls, so many of the common problems were usually resolved with complex `if`s, and switches could be resolved with virtual calls instead.

At the beginning I used also STL, which is usually much more performing than any dynamic

typing based library (STL maps are at least 10 per cent faster than PYTHON string dictionaries), but that carried on a relevant problem of interoperability with other programs. FALCON was also meant to be a scripting engine, and applications often have different ideas on which version of the STL they would like to use. Moving STL across DLLs is quite hellish and it's a deathblow to the binary compatibility of C++ modules (already less stable than C module binary compatibility by nature). Also, STL caused the code to grow quite a lot, and a lot more than I wanted; so, I temporarily switched back to dynamic typed structures, which are slower, to be sure to clear the interface across modules from external dependencies.

Recently, having found that new compilers are extremely efficient on the fast path of exception raising (actually faster than a single `if` on an error condition), I have introduced exceptions where I had cascades of controls for error being generated deep inside the VM.

In short, FALCON uses C++ where it can bring advantages in term of speed and code readability and maintainability, while still being C-oriented on several low-level aspects.

This may seem like reducing the interoperability with C applications; but this isn't the case. One of our first works as an open source project was the preparation of the *FXchat* scripting plugin for the famous *Xchat* program; as many know the *Xchat* plugin API is pure (and raw) C. Yet, the interface blends gracefully in the framework without any interoperability problems, and even without any particular inefficiency, as the FALCON scripting engine is confined in its own loadable module, and the *FXchat* module acts as a bridge. The code is even simple and direct, and it is easy to compare it against other scripting language plugins written in C that soon get much more complex than ours.

The same can be said for modules binding external libraries. We bind gracefully with both the DCOP library (written in C++) and the SDL library set (written in C), with complete interoperability and no performance or compatibility problems at all.

How is Falcon currently being adopted by developers?

Falcon is still little known on the scene, and with monsters like PYTHON and PERL around, being fed by big institutions like REBOL and ERLANG, the diffidence towards new products is great. On the other hand, it must be said that many developers who have been exposed to FALCON have been impressed by it, so much so that they didn't want to be without it anymore! Sebastian Sauer of the *Kross* project worked hard to have FALCON in *Kross* and KDE; Dennis Clarke at BlastWave is redesigning the BlastWave open source package repository Web interface with FALCON and is helping in porting all the FALCON codebase to Sun platforms – AuroraUX SunOS distro has decided to adopt it as the official scripting language (along with ADA as the preferred heavyweight development language). We receive many 'congrats' messages daily, but as we practically started yesterday (we went open source and begun getting distributed a few months ago), we have the feeling that there are many interested developers taking a peek, and staring from behind the window to see if the project gets consistent enough to ensure a stable platform for future development.

On this topic, we're starting to receive interesting proposals from some formal institutions. At the moment it's just a matter of interest and work exchange, but if things go on growing with the rhythm we've been observing recently, we'll soon need to fire up an economic entity to back the FALCON project.

What's the Falcon ecosystem like?

In the beginning, and for a few years, it was just me. I say 'just' in quotes because FALCON wasn't meant as a spare time project, but rather a professional project aiming to interact with my daily job on high-profile computing. The fact that I have been able to sell my skills and my consulting time, rather than my software, has allowed me to separate some internal projects that cannot be disclosed, from generic parts that I have shared via open source (there are also some older libraries of mine on SourceForge, which I employed on various projects). Since early 2007, some contributors have checked out the code and occasionally provided patches, but the first contributions other than mine to the repository are from 2008.

I have a community of about 10 to 12 developers, contributors and packagers actively working on the project, either externally or providing code in the core, or subsidiary modules, or on

related projects. Their contributions are still few in number and weight, but entering in a project as wide and complex as Falcon requires time. We're also preparing a sort of 'mini-SourceForge' to host Falcon-based and Falcon-related projects.

If developers want to know a bit about our project style, we are as open as an open source project can be. New ideas have been constantly integrated into our engine thanks to the help and suggestions of people either being stable in the project or just passing by and dropping a line. Although it has been impossible up to date, I am willing to pass down my knowledge to anyone willing to listen and lend a hand to the Falcon project. So, if developers are searching for a way to make a difference, stick with us and we'll make your vote to count!

Does the language have its own repository system?

No. It may seem strange being said by me, but I really don't like to reinvent the wheel. We're using SVN, but we may switch to GIT or something more Web-oriented if the need arises. In this moment, I don't consider the commit history to be very relevant (with an exception for the 0.8 and 0.9 head branches), so switching to a new repository wouldn't pose any problem.

Falcon seems to have an emphasis on speed, is this important within programming languages?

Speed is not important within programming languages – it is important for some tasks that may be solved by some programming language. As I said, I wrote HASTE out of a need for speed that wasn't addressed by any other scripting language, but Falcon evolved from HASTE for other reasons. If speed was everything, scripting languages wouldn't exist. On basic operations, the best in our class can perform about 30 times slower than C doing the same things, and that's definitely slow. Versatility, adaptability, maintainability, adequacy, integratability, complexity and many other factors play a major role in deciding which language to adopt.

Speed is the most important factor in the sense that it is the prerequisite of everything, but it's not language specific. Speed is determined by the complete 'input-processing-output' line, and what a scripting language does into that line is usually a small part. If your IPO line doesn't meet the requirement, nothing else is relevant; in the case of early Falcon, no other scripting engine was able to let my applications be used in an IPO line efficient enough to do the job on time. Once your whole system meets the IPO time requirements, speed ceases to be in the equation and everything else but speed becomes relevant. It's a binary choice: your whole IPO line is either fast enough or not.

When we say 'fast,' we mean that we concentrated our development towards helping the whole system around Falcon to use it as fast as possible. VM speed is also relevant, as there are some tasks in which you want to use heavily VM-based calculations, but it plays a secondary role, in our view, with respect to the 'service' offered to the surrounding world (applications, modules, threads) to let them run faster. This is why we have been able to live seven years without a single optimisation on the VM itself, and this is why we're starting to optimise it now, when we have completed our reflection model and serviced the surrounding world the best we can.

How does Falcon's compile-time regular expression compilation compare with the library routines of other languages?

Actually, compile-time regular expression was scheduled to be in by now (April 2009), but we went a bit long on the 0.9 release and this moved compile-time regex development a bit forward in time. However the topic is interesting, because it allows me to show three mechanisms at binding level that may be useful to the users of the scripting engine.

The plan is as follows: Falcon modular system provides to C++ a mechanism called 'service.' A service is a virtual class publishing to C++ what the module would publish to the scripts loading it. Since 0.8.12, the Falcon compiler has had a meta-compiler that fires a complete virtual machine on request. Once we accept the idea of meta-compilation, the compiler may also use the environmental settings to load the Regex module and use its methods from the service interface; that's exactly like calling the C functions directly, with just a virtual call indirection layer (which is totally irrelevant in the context of compiling a regular expression).

Since 0.9, items themselves are in charge of resolving operators through a function vector

called `item_co` (item common operations). We may either introduce a new item type for strings generated as compile time regular expressions, and provide them with an `item_co` table partially derived from the other string item type, or just create them as a string with a special marker (we already have string subtypes) and act with branches on equality/relational operators. On modern systems, a branch may cost more than a simple call in terms of CPU/memory times, so I would probably go for adding an item type (that would be also useful at script level to detect those special strings and handle them differently).

The fact that we want to use the Regex module at compile time is another interesting point for embedders. If we included regular expressions in the main engine, we would grow it some more and we would prevent the embedders from the ability of disabling this feature.

One of the reasons I wanted FALCON was to allow foreign, less-trusted scripts to be compiled remotely and sent in pre-compiled format to a server for remote execution. The executing server may want to disable some features for security reasons (it may forbid to use file i/o), and that just on some unprivileged VM, while the administrative scripts run at full power. That was impossible to do with the other scripting engines unless there were deep rewrites. FALCON modular system allows the modules to be inspected and modified by the application prior to injection into the requesting VMs. So, a server or application with differently privileged script areas can pre-load and re-configure the modules it wishes the script to be able to use, preventing the loading of other modules, while letting privileged scripts to run unhindered.

Regexes are heavy, and not all embedding applications may wish their scripts to use them. For example, a MMORPG application may decide that AI bots have no use for regular expressions, and avoid providing the Regex module. At this point, the compiler would simply raise an error if it finds a `r"..."` string in a source, and the VM would raise an error if it has to deal with a pre-compiled Regex in a module. At the same time, as the Regex module is mandatory on any complete command line FALCON installation, command line scripts can use Regexes at the sole extra cost of dynamic load of the Regex module, which is irrelevant on a complete FALCON application, and that would be cached on repeated usage patterns as with the Web server modules.

Do you plan to develop your own Regex library to drive your regular expressions?

No, we're happy with PCRE, which is the best library around in our opinion, and even if it's relatively huge, having it in a separate module loaded on need seems the way to go. We keep updated as possible with its development, providing native binding on some systems where PCRE is available (many Linux distributions) and shipping it directly in the module code where it is not available.

Is the embeddable aspect of Falcon versatile?

I talked diffusely about that in the Regex example above, but other than the reconfigurability and sharing of pre-loaded modules across application VM, we have more. The VM itself has many virtual methods that can be overloaded by the target application, and is light enough to allow a one-vm-per-script model. Heavy scripts can have their own VM in the target application, and can happily be run each in its own thread; yet VMs can be recycled by de-linking just run scripts and linking new ones, keeping the existing modules so that they're already served to scripts needing them.

The VM itself can interact with the embedding application through periodic callbacks and sleep requests. For example, a flag can be set so that every sleep request in which the VM cannot swap in a coroutine ready to run is passed to the calling application that can decide to use the idle time as it thinks best. For instance, this allows spectacular quasi-parallel effects in the *FXChat* binding, where the `sleep()` function allows *Xchat* to proceed. This may seem a secondary aspect, but other engines are actually very closed on this; once you start a script or issue a callback, all that the application can do is to hope that it ends soon. With FALCON you can interrupt the target VM with simple requests that will be honoured as soon as possible, and eventually resume it from the point it was suspended and inspected.

Since 0.9 we have introduced even a personalized object model. FALCON instances need not be full blown FALCON objects; the application may provide its own mapping from data to items

travelling through the FALCON VM. Compare this with the need of creating a dictionary at each new instance, and having to map each property to a function retrieving data from the host program or from the binded library.

Other classes which you can override are the module loader, which may provide FALCON modules from other type of sources, or from internal storage in embedding applications, and since 0.9 the URI providers. Modules and embedding applications can register their own URI providers, so that opening a module in the `app://` space would turn into a request to get a module from an internally provided resource, or opening a stream from a script from `app://` would make possible to communicate binary data via streams to other parts of the application.

Frankly, we did our best to make our engine the most versatile around. They say LUA is very versatile, as you can reprogram it as you wish. But then, that is true for any open source project.

How will the new multithreading design in version 0.9 innovate the scripting language panorama?

There are two good reasons why multithreading in scripting languages are delicate matters (that many didn't even want to face). The first is that multithreading can break things. In 'good multithreading' (multithreading which is allowed to actually exploit parallel computational power of modern architectures without excessive overhead), there is no way to recover from an error in a thread. A failing thread is a failing application, and that is a bit problematic to be framed in the controlled execution concept behind scripting language virtual machines.

The second reason is, as LUA developers point out, that a language where `a = 0` is not deterministic cannot be proficiently used in multithreading. Some scripting languages make `a = 0` be deterministic and visible across threads by locking every assignment instruction, and that is a performance killer under many aspects. It doesn't only deplete performance on the script itself, but in case of concurrent programming in an application, it may severely deplete the host application performance by forcing it to unneeded context switches.

We opted for a pure agent-based threading model. Each thread runs a separate virtual machine, and communication across threads can happen only through specialised data structures. In this way, each virtual machine can run totally unhindered by global synchronisation. It is possible to share raw memory areas via the `MemBuf` item type, or to send complete objects created upon separate elaboration via a interthread item queue.

The point is that, in our opinion, multithreading in scripting languages cannot be seen as multithreading in low-level languages, where each operation can be mapped to activities in the underlying parallel CPUs. The idea of 'mutex/event'-based parallel programming is to be rejected in super high-level languages as scripting languages, as there are too many basic operations involved in the simplest instruction. Since, in complex applications written even with low-level languages, those primitives are used by law to create higher-level communication mechanisms, our view is that multithreading in scripting languages should provide exactly those mechanisms, without trying to force the scripts to do what they cannot proficiently do, that is, low-level synchronization primitives.

When I write a server, I find myself struggling to create complex synchronisation rules and structures through those primitives, avoiding to use them directly, and I don't see why we should bestow the same struggle on script users. The realm where primitive synchronisation is useful is not a realm where scripting languages should play a direct role – it's where you would want to write a C module to be used from the scripting language anyhow.

In 0.9 we have introduced an inter-thread garbage collector that accounts for objects present in more virtual machines. This is already exploited via the sharing of `MemBuf` instances, but we plan to extend this support to other kind of objects. For example, it is currently possible to send a copy of a local object to another thread via an item queue (the underlying data, possibly coming from a module or from an embedding application, can actually be shared; it's just the representation each VM has of the object that must be copied). This makes it a bit difficult to cooperate on creating complete objects across threads, and even if this works in term of agent-based threading, we're planning to use the new interthread GC system to be able to send deep items across threads. Since 0.9, it is already possible to create deep data externally (i. e.

in the embedding application or in a module) and send it to a VM in a different thread.

The only problem left in doing it natively across two different VMs is ensuring that the source VM won't be allowed to work on the object and on any of the data inside it while the target VM is working on it. Even if this may seem a limitation, it's exactly what the 'object monitor' approach to multithreading dictates, and it is perfectly coherent with our view of higher-level parallel abstraction. Version 0.9 also introduces the mechanism of interthread broadcasts, with message oriented programming extended to interthread operations. We still have to work that out completely, but that's the reason why we're numbering this release range '0.9.'

Finally, as the VM has native multithread constructs now, we may also drop the necessity to have different VMs for different threads, as each thread may just operate on its own local context, while common operations on the VM (as loading new modules) can be easily protected. Still, we need to consider the possibility of multiple VMs living in different threads, as this is a useful model for embedding applications.

How can a software developer get into Falcon development?

Easily. We can divide the support you may give to Falcon in mainly five areas. I rank them by order of weighted urgency/complexity ratio.

1. **Modules**. We need to extend the available features of the language, and modules are a good place from where to start, both because they are relatively simple to write and build and because they put the writer in contact with the VM and item API quite directly. At the moment we don't have a comprehensive module writer's guide, but examples are numerous and well commented, and the API of both the VM and items are extensively documented. A skeleton module is available for download from our 'extensions' area on the site, and provides an easy kick-off for new projects. Some of the most wanted modules and bindings are listed.

2. **Applications**. We'd welcome some killer application as a comprehensive CMS written in Falcon, but even simpler applications are welcome.

3. **Extensions and embeddings**. As a scripting engine, we welcome people willing to drive their applications with Falcon. For example, the binding with Kross into KDE applications. We have a cute scripting engine binding for *XChat*, and we'd like to have for other scriptable applications (other IM systems, editors, music players etc). We need also to extend the existing HTTP server module binding engine and to apply it to more servers. At the moment we only support Apache.

4. **Falcon core**. Maintaining and extending the core system, the core module and the *Feathers* is still quite challenging: the 0.9 development branch has just started and we need to exploit the most advanced techniques in terms of memory management and compiler optimisations existing around, or find new ones. We'll introduce at least two more paradigms in this round; logic programming and type contract programming, and there's plenty of work to do on tabular programming. The area is still open, so if you really want to get the hands dirty on the top-level technology in the field, this is the right place and the right time to give a try at that.

5. **IDE**. We need an IDE for development and debugging of Falcon applications. A terribly interesting tool would be an embeddable IDE that applications may fire up internally to manage their own scripts (consider game mod applications, but also specialised data-mining tools). Falcon has a quite open engine, and integrating it directly into the environment shall be easy. I put it for fifth as an IDE is useless if the language doesn't develop the other four points in the meanwhile, but having an IDE ready when the other four points will be satisfactorily advanced would be really a godsend.

Jumping in is easy – just get the code you want to work on from our SVN (or make a complete installation of Falcon + dev files and install the skeleton module if you want to write your own extension) and do something. Then give us a voice through our newsgroup, mail or IRC, and

you're in. Developers may join as contributors and enter the Committee if their contribution is constant and useful.

Have you faced any hard decisions in maintaining the language?

Yes and no. Yes in the sense that there have been many no-way-back points, and so the decisions were hard at those spots where I had to choose to do one thing rather than another. For example, when I decided to drop the support for stateful functions, a cute feature of the former FALCON language which was used to build stateful machines. Stateful machines were quite useful in many contexts, and having language constructs directly supporting them was interesting. But we observed that the cost of entering and exiting *every* function was too high due to the need to check if it was a stateful function or not, and this lead to abandoning those constructs. So, while this kind of decisions were hard in terms of 'harness (metallurgy),' none of the decisions I made was difficult to take.

Every decision was taken after deep technical cost-benefit analysis, the more the 'hardness (metallurgy),' the deeper the analysis. So, with a solid base on which to decide, and having hard evidence and data on which to work on, every decision was actually easy, as it was the only viable one or the best under technical aspects.

Looking back, is there anything you would change in the language's development?

I would have made more of an effort to go open source sooner. The market was already quite full by the time I started, so I was a bit shy in exposing the results of my research to the public until proving that my solution was technically better in their specific application field. But this slowed down the development of subsidiary areas, like the modules. Developers may have been attracted not just by a better approach to some problem, but just by the idea of doing something fun with a new tool. I underestimated this hedonistic aspect of open source, and now I am a bit short of breath having to take care of the inner stuff alone. This is why I am so eager to pass my knowledge around and help anyone willing to carry on the project.

Where do you envisage Falcon's future lying?

In being a good scripting language. For how tautological it may seem, this is not really the case. Many other languages, even the most prominent ones, have overgrown their scope and now are trying to invade areas that were not exactly meant for untyped, ultra-high-level, logic-oriented scripting languages. If it's true that one must find new limits, and break them, it's also true that there's pre-determination in nature. From a peach seed you will grow a peach, and a planet can be many things, but never a star.

By overdoing their design, they're not evolving, they're just diluting their potential. Our aim is to provide an ever growing potential of high-level control logic and design abstraction, at disposal of both application in need of a flexible inner variable logic engine, or directly at the command of the developers; this, at an affordable cost in terms of performance (not with respect to other scripting languages, but with respect of doing things the hard-coded way).

Forth: Charles Moore

Charles H. Moore invented Forth *while at the US National Radio Astronomy Observatory to help control radio telescopes and data-collection/reduction systems. Here he chats about why* Forth *was invented, as well as why he still works with* Forth *today*

How did Forth come into existence?

Forth came about when I was faced with a just-released IBM 1130 minicomputer. Compiling a Fortran program was a cumbersome procedure involving multiple card decks. I used Fortran to develop the first Forth, which could use the disk and graphics display that Fortran couldn't. Because it was interactive, my programming was much faster and easier.

Was there a particular problem you were trying to solve?

This computer was at Mohasco Industries. Their main product was carpet and the problem was to determine if the 1130 could help design carpets. Forth was great for programming the 2250 display, but ultimately the lack of color doomed the project.

Did you face any hard decisions in the development of the language?

The hardest decision in developing Forth was whether to do it. Languages were not casually designed. It was reputed to require a brilliant team and man-years of effort. I had my eye on something quick and simple.

How did Forth get its name?

I decided to call it Fourth, as in 4th-generation computer language. But the 1130 file system limited names to 5 characters, so I shortened it to Forth. A fortuitous choice, since Forth has many positive associations.

I have read that Forth was developed from your own personal programming system, which you began to develop in 1958. Can you tell us a little more about this?

My personal programming system was a deck of punch cards, [now] sadly lost. It had a number of Fortran subroutines that did unformatted input/output, arithmetic algorithms and a simple interpreter. It let me customize a program via its input at a time when recompiling was slow and difficult.

Why did you incorporate Reverse Polish notation into the language?

Reverse Polish notation is the simplest way to describe arithmetic expressions. That's how you learn arithmetic in grade school, before advancing to infix notation with Algebra. I've always favored simplicity in the interest of getting the job done.

Was the language developed particularly for your work at the National Radio Astronomy Observatory?

I did most of my work at NRAO in Forth: controlling several radio telescopes and data-collection/reduction systems, with the reluctant approval of the administration. The only reason this was permitted was that it worked: projects took weeks instead of years, with unparalleled performance and capabilities.

If you had the chance to re-design the language now, would you do anything differently?

Would I do anything differently? No. It worked out better than I dreamed. The thing about Forth is that if I wanted a change, I made it. That's still true today. Forth is really a language tool kit. You select and modify every time you encounter a new application.

Do you still use/work with Forth?

Yes indeed, I write Forth code every day. It is a joy to write a few simple words and solve a problem. As brain exercise it far surpasses cards, crosswords or Sudoku; and is useful.

What is your reaction to comments such as the following from Wikipedia: '*Forth is*

a simple yet extensible language; its modularity and extensibility permit the writing of high-level programs such as CAD systems. However, extensibility also helps poor programmers to write incomprehensible code, which has given Forth a reputation as a "write-only language"'?

All computer languages are write-only. From time to time I have to read C programs. They are almost incomprehensible. It's not just the syntax of the language. It's all the unstated assumptions. And the context of the operating system and library. I think FORTH is less bad in this regard because it's compact; less verbiage to wade thru. I like the observation that FORTH is an amplifier: a good programmer can write a great program; a bad programmer a terrible one. I feel no need to cater to bad programmers.

Do you know of many programs written using Forth, and if so, what's your favourite?

FORTH has been used in thousands of applications. I know of very few. The FORTH Interest Group held conferences in which applications were described. The variety was amazing. My current favorite is that FORTH is orbiting Saturn on the Cassini spacecraft.

In your opinion, what lasting legacy do you think Forth has brought to the Web?

The various Web pages and forums about FORTH make a powerful point: FORTH is alive and well and offers simple solutions to hard problems. FORTH is an existence proof. A lasting legacy to KISS (keep it simple, stupid).

What made you develop colorForth?

I was driven away from FORTH by the ANSI standard. It codified a view of FORTH that I disliked: megaFORTH; large, unwieldy systems. I was finally faced with the need for VLSI chip design tools. And [I was also] blessed with some insight as to how FORTH could be made faster, simpler and more versatile. Hence, COLORFORTH. Sadly [it has been] ignored by most FORTH programmers.

Are there many differences between Forth and colorForth?

COLORFORTH adds a new time to FORTH. FORTH is intrinsically interactive. The programmer must distinguish compile-time from run-time, and switch back-and-forth between them. Anything that can be done at compile-time will save run-time. In COLORFORTH there is also edit-time, which can save compile-time. The COLORFORTH editor pre-parses text into Shannon-coded strings that are factored into 32-bit words. Each word has a 4-bit tag the compiler uses to interpret it. Compilation is very fast. COLORFORTH also restricts its primitives so they can be efficiently executed by a FORTH chip.

Where do you envisage Forth's future lying?

I'm betting that parallel computers will be the future, and FORTH is an excellent parallel-programming language. But I expect that conventional languages will become more complex in order to describe parallel processes. Computer scientists must exercise their ingenuity and have something non-trivial to teach.

Do you have any advice for up-and-coming programmers?

I think it behooves new programmers to sample all the languages available. FORTH is the only one that's fun. The satisfaction of finding a neat representation cannot be equaled in FORTRAN, C or even LISP. (And mentioning those languages surely dates me.) Try it, you'll like it.

What are you working on now?

Currently I'm working with OKAD, my COLORFORTH CAD tools, to design multi-core computer chips. They're small, fast and low-power, just like FORTH.

Would you like to add anything else?

To reiterate: FORTH is an existence proof. It shows that a computer language can be simple and powerful. It also shows that 'The race is not to the swift.' The best solution is not necessarily the popular one. But popularity is not a requirement. There are many applications where a good solution is more important than popular methodology.

Groovy: Guillaume Laforge

Groovy's Project Manager, Guillaume Laforge, tells the development story behind the language and why he thinks it is grooving its way into enterprises around the world. Groovy, he says, is ultimately a glue that makes life easier for developers – and it has nothing to do with Jazz.

How did you come up with the name Groovy? Is it a reference to counter culture or are you a jazz fan?

There's a little known story about the invention of the name!

Back in the day, in 2003, after suffering with Java and loving the features available in dynamic languages like Ruby, Python and Smalltalk, lots of questions arose of the form of, 'Wouldn't it be "groovy" if Java had this or that feature and you could replace said feature with closures, properties, metaprogramming capabilities, relaxed Java syntax?' and more.

When it came to choosing a name, it was obvious that a new language with all those great features would have to be called 'Groovy'! So it's not really a reference to counter culture, nor about jazz, but just about the dream of having a language close enough to Java, but more powerful and expressive. That's how Groovy came to life.

What are the main differences between Groovy and other well-known dynamic languages like Ruby, Python and Perl?

The key differentiator is the seamless integration with the Java platform underneath. It's something no other languages provide, even alternative languages for the JVM (Java Virtual Machine), or at least not up to the level that Groovy does.

First of all, the grammar of the language is derived from the Java 5 grammar, so any Java developer is also a Groovy developer in the sense that the basic syntax is already something he would know by heart. But obviously Groovy provides various syntax sugar elements beyond the Java grammar. The nice aspect of this close relationship is that the learning curve for a Java developer is really minimal.

Even at the level of the APIs, aspects such as the object orientation and the security model are all just what you would be accustomed to with Java. There's really no impedance mismatch between Groovy and Java. That's why lots of projects integrate Groovy, or why companies adopt the Grails web framework.

What led you to develop Groovy – was it to solve a particular problem or carry out a particular function that you could not do in another language?

Back in 2003, I was working on project that was a kind of application generator where there was a Swing designer User Interface (UI) to define a meta-model of the application you wanted to build, and you could define the tables, columns, and UI widgets to represent the data and layout. This meta-model was deployed on a web application that interpreted that model to render a live running application. It was a pretty powerful system.

The project also allowed some customised UI widgets to render certain fields, like autocomplete fields and such, and you could develop your own widgets. But those widgets had to be developed in Java, compiled into bytecode, archived in a JAR file, and – the biggest drawback of all – you then had to deploy a new version of the web application to take this new widget into account.

The obvious problem was that all the customers using those generated applications had to stop using them for a moment, for a maintenance window, so that only one customer could have access to that new widget he needed. It was at that point that I decided a kind of scripting language would be useful to develop those widgets, and have them stored in the meta-model of the applications, and interpreted live in the running server.

What was the primary design goal for the language?

Groovy's design goal has always been to simplify the life of developers.

We borrowed interesting features from other languages to make Groovy more powerful, but

have [always had a] strong focus on a total seamless integration with Java. Because of these goals, Groovy is often used as a superglue for binding, wiring, or configuring various application components together. When we created the language, this glue aspect was clearly one of the primary functions.

How is it most often used?

Companies using Groovy usually don't write full applications in Groovy, but rather mix Groovy and Java together. So Groovy is often used as a glue language for binding parts of applications together, as a language for plugins or extension points, as a more expressive way to create unit and functional tests, or as a business language. It's very well suited for defining business rules in the form of a Domain-Specific Language.

How widely is Groovy being used and where?

Groovy is very often the language of choice when people need to integrate and use an additional language in their applications, and we know of lots of mission-critical applications are relying on Groovy.

For instance, Groovy is often used in financial applications for its expressivity and readability for writing business rules, and also because if its usage of BigDecimal arithmetics by default which allows people to do exact calculations on big amounts of money without important rounding errors. For example, there is a big insurance company in the US that used Groovy for writing all its insurance policy risk calculation engine. There is also a European loan granting platform working with 10 per cent of all the European banks, dealing with one billion Euros worth of loans every month, which uses Groovy for the business rules for granting loans and as the glue for working with web services.

The financial sector is not the sole one: Groovy is also being used by biomedical and genetics researchers, by CAD software and more.

How many developers work on Groovy?

We currently have two full-time persons working on Groovy, plus a handful of super-active committers. We've got a second-tier of casual committers who focus on particular areas of the project. Groovy is a very active project that has seen a long list of committers and contributors over the course of its development.

Can you tell us a bit more about Grails (formerly Groovy on Rails) and is it, in your opinion, a good example of what can be done with Groovy?

Grails is a highly productive web development stack. More than a mere Web framework, it provides an advanced integration of the best-of-breed open source software (OSS) components, such as Spring and Hibernate, to provide a very nice experience for developers using it, while also taking care of various other aspects like the project build, the persistence, a rich view layer and an extensible plugin system. Clearly, Grails leverages Groovy heavily, to bring productivity boosts to developers at every step of the project. Grails' choice of Groovy and all the other components it uses makes it a very compelling platform for high-traffic and complex applications.

What are some other interesting open source applications using Groovy?

Griffon is based on the Grails foundations and extends Groovy's own Swing builder system to let you create complex and rich desktop applications in Swing. Griffon is really to Swing development what Grails is for Web development.

In the testing space, Easyb brings developers a DSL for Behavior-Driven-Development testing, and Spock provides some advanced testing and mocking techniques to unit testing. Let me also mention Gradle, which is a very nice and advanced build system.

What are the biggest tasks you are currently working on with the language development?

We always have two ongoing efforts at the same time: maintaining and evolving the current stable branch, as well as working and innovating on the development branch.

For instance, we've just released a minor version of Groovy 1.6 which solves some bugs and has some minor enhancements, and we have also just released a preview of the upcoming

Groovy 1.7 full of new features. Groovy 1.7 will make it easier for extending the language through compile-time metaprogramming capabilities. It will also provide better assertion messages for unit tests, the ability to use annotations in more places in your programs and lots more.

Why did you choose an Apache License over other free and /or open licences?

We felt that the Apache License was a great and open licence to use, so that anybody is free to embed, reuse, or even fork the language in whatever they see fit for their own needs, and integrate it in their own applications. The choice was also easy with some of the original committers coming from the Apache Software Foundation.

As it is in some ways a superset of Java, it would be easy for Java developers to learn, but what is the learning curve for developers without a Java background?

Of course Groovy is easy to learn for Java developers, but thanks to its 'scripting' aspects, it's still rather easy to learn for users coming from a different background.

As long as you're somewhat familiar with a language with a C-like syntax, it's simple to comprehend. There are of course some APIs to learn, as with any language and platform, but you can learn them as you need them without any upfront cost of learning. So even without a Java background, the learning curve isn't that stiff.

What is your favourite Groovy feature?

This is a tricky question! There are really many aspects of the language that I love!

I guess if I had to choose just one, that would be Groovy's support for closures. With closures, you can start thinking differently about how you solve your everyday problems, or create complex algorithms. Closures give you an additional layer of abstraction for encapsulating code and behaviour, and even data (thanks to Groovy builders). Also, with various helper methods added to Java collections, in combination with closures, you've got the power of functional languages at your disposal.

What has been the greatest challenge in developing Groovy and how did you work around this?

I would say that the two main challenges have been about a total seamless interoperability and integration with Java, as well as performance.

The former has always been part of our design goals, so we've always done our best to take care of all the bugs and keep up with the pace of Java itself (for instance when Java 5 introduced annotations, enums, and generics).

For the latter, we made sure that Groovy would be the fastest dynamic language available (both in and outside of the JVM). We used various techniques, such as 'call site caches' and related techniques. We're also very enthusiastic and optimistic about the upcoming JSR-292 'invokedynamic' bytecode instructions coming soon into the Java Virtual Machine, which should bring very significant performance boosts.

Do developers in corporate environments have trouble using non-standadised and relatively new languages like Groovy in the workplace?

It depends, [but this can happen] in some cases. Groovy is an easy sell, as after all it's just another library to put on the classpath, and in some other cases it's more problematic as certain companies are really strict and avoid adding any other dependency in their projects, trying to mitigate any additional risk. Usually though, the various advantages Groovy brings help sell it to more conservative organisations.

Until recently, the tooling wasn't ideal either, but JetBrains with their incredible Groovy and Grails support in IntelliJ IDEA paved the way. We also have great support in NetBeans, and thanks to the SpringSource Eclipse team, the Eclipse plugin for Groovy is going to progressively catch up with the competitors. Groovy is now a much easier sell than it was a few years ago and a lot of companies trust Groovy for their advanced needs.

A Slashdot reader has said in a post months ago that Groovy is poised to convert the enterprise crowd. Do you agree with this statement?

More and more companies are relying on Groovy for doing business – even critical apps dealing with large amounts of money. So clearly, Groovy is now a key asset to such companies and businesses. And the fact Groovy is very easy to learn and use, and is so well integrated with Java, makes it a nice fit for bringing more agility and power in your applications.

Where do you see Groovy heading in the future?

This is a very good question! After each major release, we're wondering whether we will be able to add some new innovative and useful features to the language. And in the end, we always find something!

As I mentioned already, there are areas where we continue to innovate, like our compile-time metaprogramming techniques and our extended annotation support.

We're also considering certain features we find interesting in other languages and their respective APIs, for instance Erlang's actor concurrency model, pattern matching like in functional languages such as OCaml, or parser combinators from Haskell.

We always try to find new features that bring real value and benefits to our users.

Haskell: Simon Peyton-Jones

We chat with Simon Peyton-Jones about the development of Haskell. *Peyton-Jones is particularly interested in the design, implementation, and application of lazy functional languages, and speaks in detail of his desire to 'do one thing well,' as well as his current research projects being undertaken at Microsoft Research in Cambridge, UK.*

Was Haskell created simply as an open standard for purely functional programming languages?

Haskell isn't a standard in the ISO standard sense – it's not formally standardized at all. It started as a group of people each wanting to use a common language, rather than having their own languages that were different in minor ways. So if that's an open standard, then yes, that's what we were trying to do.

In the late 1980s, we formed a committee, and we invited all of the relevant researchers in the world, as at that stage the project was purely academic. There were no companies using lazy functional programming, or at least not many of them. We invited all of the researchers we knew who were working on basic functional programming to join in.

Most of the researchers we approached said yes; I think at that stage probably the only one who said no was David Turner, who had a language called Miranda, and Rinus Plasmeijer, who had a language called Clean. He was initially in the committee but he then dropped out. The committee was entirely by consensus – there wasn't a mechanism whereby any one person decided who should be in and who should be out. Anybody who wanted could join.

How did the name come about?

We sat in a room which had a big blackboard where we all wrote down what we thought could be possible candidates for names. We then all crossed out the names that we didn't like. By the time we were finished we didn't have many!

Do you remember any of the names that you threw up there?

I'm sure there was Fun and Curry. Curry was Haskell Curry's last name. He'd already given his name to a process called 'currying' and we ended up using Haskell instead of Curry, as we thought that there were too many jokes you could end up making about it!

So what made you settle on Haskell?

It was kind of a process of elimination really, and we liked that it was distinctively different. Paul Hudak went to see Curry's widow who kindly gave us permission to use his name. The only disadvantage is that people can think you mean 'Pascal' rather than 'Haskell.' It depends on the pronunciation – and it doesn't take long to de-confuse people.

Did you come across any big problems in the early stages of development?

The Haskell project was meant to gather together a consensus that we thought existed about lazy functional programming languages. There weren't any major issues about anything much, as we had previously agreed on the main issues and focus. There were also some things that we deliberately decided not to tackle: notably modules. Haskell has a basic module system but it's not a state of the art module system.

Why did you decide not to tackle this?

Because it's complicated and we wanted to solve one problem well, rather than three problems badly. We thought for the bits that weren't the main focus, we'd do something straightforward that was known to work, even if it wasn't as sophisticated as it could get. You only have so much brain capacity when you're designing a language, and you have to use it – you only have so much oxygen to get to the top of the mountain. If you spend it on too many things, you don't get to the top!

Were the modules the main elements you decided not to tackle, or were there other elements you also avoided?

Another more minor thing was records. HASKELL has a very simple record system, and there are lots of more complicated record systems about. It's a rather complicated design space. Record systems are a place where there's a lot of variation and it's hard to know which is best. So again, we chose something simple.

People sometimes complain and say 'I want to do this obviously sensible thing, and HASKELL doesn't let me do it.' And we have to say, well, that was a place we chose not to invest effort in. It's usually not fundamental however, in that you can get around it in some other way. So I'm not unhappy with that. It was the economic choice that we made.

So you still support these decisions now?

Yes. I think the record limitation would probably be the easiest thing to overcome now, but at this stage HASKELL is so widely used that it would likely be rather difficult to add a complete record system. And there still isn't an obvious winner! Even if you asked 'today, what should records in HASKELL look like?,' there still isn't an obvious answer.

Do you think that new modules and record formats will ever be added on to the language?

You could certainly add an ML style module system to HASKELL, and there have been a couple of papers about that in the research literature. It would make the whole language significantly more complicated, but you'd get some significant benefits from it. I think at the moment, depending on who you speak to, for some people it would be the most pressing issue with HASKELL, whereas for others, it wouldn't.

At the moment I don't know anyone who's actively working on a revision of HASKELL with a full-scale module implementation system.

Do you think that this is likely to happen?

I doubt it will be done by fitting it [new modules and record formats] on to HASKELL. It might be done by a successor language to both ML and HASKELL, however.

I believe that a substantial new thing like modules is unlikely. Because HASKELL's already quite complicated right now, adding a new complicated thing to an already complicated language is going to be hard work! And then people will say, 'oh, so you implemented a module system on HASKELL. Very well, what's next?' In terms of academic brownie points, you don't get many unfortunately.

In 2006 the process of finding a new standard to replace Haskell 1998 was begun. Where is this at now? What changes are being made?

HASKELL'98 is like a checkpoint, or a frozen language specification. So HASKELL itself, in various forms, has continued to evolve, but if you say HASKELL'98, everyone knows what you mean. If you say HASKELL, you may mean a variety of things.

Why did the '98 version get frozen in particular?

Because people had started saying that they wanted to write books about HASKELL and that they wanted to teach it. We therefore decided to freeze a version that could be relied on, and that compiler writers like me can guarantee to continue to maintain. So if you have a HASKELL'98 program it should still work in 10 years time.

When we decided to do it, HASKELL'98 was what we decided to call it. Of course, 5 years later we may have done something different. That's what's happening now, as people are saying 'I want to use language features that are not in HASKELL'98, but I also want the stability that comes from a 'branded' or kite marked language design – the kind that says this isn't going to change and compilers will continue to support it.'

So it's an informal standardization exercise again – there's no international committees, there's no formal voting. It's not like a C++ standard which is a much more complicated thing.

The latest version is called *Haskell Prime* (*Haskell'*) at the moment. It's not really a name, just a placeholder to say that we haven't really thought of a name yet!

So how is Haskell Prime developing?

Designing a whole language specification, and formalizing it to a certain extent, or writing it

down, is a lot of work. And at the moment I think we're stalled on the fact that it's not a high enough priority for enough people to do that work. So it's moving rather slowly – that's the bottom line.

I'm not very stressed out about that, however. I think that when we get to the point where people care enough about having a painstaking language design that they can rely on, then they'll start to put more effort in and there'll be an existing design process and a set of choices all laid out for them. I don't see that [the current slow progress] as a failure; I see that as evidence of a lack of strong enough demand. Maybe what's there is doing OK at the moment.

One way that this has come about, is that the compiler I am responsible for (the GHC or Glasgow Haskell Compiler), has become the de facto standard. There are lots of people using that, so if you use GHC then your program will work.

I don't think that's a good thing in principle, however, for a language to be defined by an implementation. HASKELL is based on whatever GHC accepts right now, but it [HASKELL] should have an independent definition. So I would like to see *Haskell Prime* happen because I think it's healthy to see an independent definition of the language rather than for it to be defined by a de facto standard of a particular compiler.

Do you think *Haskell Prime* will eventually reach that point?

I don't know. It's a question of whether the urgency for doing that rises before somebody comes out with something startlingly new that overtakes it by obsoleting the whole language.

Have you seen anything out there that looks like doing this yet?

Not yet, no.

Are you expecting to?

It's hard to say. In my experience, languages almost always come out of the blue. I vividly remember before JAVA arrived (I was still working on HASKELL then), and I was thinking that you could never break C++'s strangle-hold on mainstream programming. And then JAVA arrived, and it broke C++'s strangle-hold!

When JAVA came, nobody provided commentary about this upcoming and promising language, it just kind of burst upon the scene. And PYTHON has similarly become extremely popular, and PERL before it, without anybody strategically saying that this is going to be the next big thing. It just kind of arrived and lots of people started using it, much like *Ruby on Rails*. There are lots and lots of programming languages, and I'm no expert [on predicting what will be big next]. I don't think anybody's an expert on predicting what will become the next big thing.

So why am I saying that? Well, it's because to supplant established languages, even in the functional programming area, like HASKELL or ML or SCHEME, you have to build a language that's not only intriguing and interesting, and enables people to write programs faster, but you also need an implementation that can handle full scale applications and has lots of libraries and can handle profilers and debuggers and graphical analyzers ... there's a whole eco-system that goes with a programming language, and it's jolly hard work building that up. What that means is that it's quite difficult to supplant that existing sort of base.

I think if you thought about it in the abstract you probably could design a language with the features of HASKELL and ML in a successor language, but it's not clear that anybody's going to do that, because they'd have to persuade all of those people who have got a big investment in the existing languages to jump ship. I don't know when something fantastic enough to make people do that jumping will appear. I don't think it's happening yet, and I don't think it's likely to happen by somebody saying that 'I've decided to do it!' but rather more organically.

Speaking of the evolution of Haskell, what do you think of variants such as Parallel, Eager and Distributed Haskell, and even Concurrent Clean?

This is all good stuff. This is what HASKELL is for. HASKELL specifically encourages diversity. By calling HASKELL'98 that name instead of HASKELL, we leave the HASKELL brand name free to be applied to lots of things. Anything that has 'Haskell' in the name is usually pretty close; it's usually an extension of HASKELL'98. I don't know anything that's called 'something-Haskell'

and that doesn't include Haskell'98 at least.

These aren't just random languages that happen to be branded, like JavaScript which has nothing to do with Java. They [JavaScript] just piggy-backed on the name. They thought if it was Java, it must be good!

Do you find yourself using any of these languages?

Yes. Concurrent Haskell is implemented in GHC, so if you say I'm using Concurrent Haskell you're more or less saying you're using GHC with the concurrency extension Data Parallel. Haskell is also being implemented in GHC, so many of these things are actually all implemented in the same compiler, and are all simultaneously usable. They're not distinct implementations.

Distributed Haskell is a fork of GHC. Some older versions run on multiple computers connected only to the Internet. It started life as being only part of GHC, but you can't use it at the same time as the concurrency extensions, or a lot of the things that are new in GHC, because Distributed Haskell is a 'fork.' It started life as the same code base but it has diverged since then. You can't take all of the changes that have been made in GHC and apply them to the distributed branch of the fork – that wouldn't work.

Concurrrent Clean on the other hand is completely different. It's a completely different language; it's got a completely different implementation. It existed before Haskell did and there's a whole different team responsible, led by Rinus Plasmeijer. At one stage I hoped that we would be able to unify Haskell and Clean, but that didn't happen. Clean's a very interesting and good language. There's lots of interesting stuff in there.

When did you think that the two might combine?

When we first started, most of us [the Haskell committee] had small prototype languages in which we hadn't yet invested very much, so we were happy to give them all up to create one language. I think Rinus had more invested in Concurrent Clean, however, and so chose not to [participate in Haskell]. I have no qualms with that, as diversity is good and we don't want a mono-culture, as then you don't learn as much.

Clean has one completely distinct feature which is not in Haskell at all, which is called uniqueness typing. This is something that would have been quite difficult to merge into Haskell. So there was a good reason for keeping two languages ... It's another thing like modules that would have been a big change. We would have had a lot of ramifications and it's not clear that it would have been possible to persuade all of the Haskell participants that the ramifications were worth paying for. It's the 'do one thing well,' again.

That sounds like the language's aim: do one thing and do it well ...

Yes. That's right.

We're seeing an increase in distributed programming for things like multi-core CPUs and clusters. How do you feel Haskell is placed to deal with those changes?

I think Haskell in particular, but purely functional programming in general, is blazing the trail for this. I've got a whole one hour talk about the challenge of effects – which in this case actually means side effects. Side effects are things like doing input/output, or just writing to a mutable memory location, or changing the value of the memory location.

In a normal language, like Pascal or Perl, what you do all the time is say 'assign value 3 to x,' so if x had the value of 4 before, it has the value of 3 now. So that these locations, called x, y & z, are the names of a location that can contain a value that can change over time.

In a purely functional language, x is the name of a value that never changes. If you call a procedure in Perl or Pascal, you might not pass the procedure any arguments and you may not get any results, but the act of calling the procedure can cause it to write to disk, or to change the values of many other variables. If you look at this call to the procedure, it looks innocuous enough, but it has side effects that you don't see. These aren't visible in the call, but there are many effects of calling the procedure, which is why you called it. If there were no effects you wouldn't call it at all.

In a functional language, if you call a function f, you give it some arguments and it returns a result. All it does is consume the arguments and deliver the result. That's all it does – it doesn't

create any other side effects. And that makes it really good for parallel evaluation in a parallel machine. Say if you call f of x and then you add that result to g of y in a functional language, since f doesn't have any side effects and g doesn't have any side effects, you can evaluate the calls in parallel.

But in a conventional mainstream programming language, f might have a side effect on a variable that g reads. f might write a variable behind the scenes that g looks at. It therefore makes a difference whether you call f and then g or g and then f. And you certainly can't call them at the same time!

It's actually really simple. If the functions that you call do not have any side effects behind the scenes, if all they do is compute a value from the input values that you give them, then if you have two such things, you can clearly do them both at the same time. And that's purely functional programming. Mainstream languages are, by default, dangerous for parallel evaluation. And purely functional languages are by default fine at parallel evaluation.

Functional, whether lazy or non-lazy, means no side effect. It doesn't mess about behind the scenes – it doesn't launch the missiles, it doesn't write to the disk. So the message of the presentation I mentioned before is that purely functional programming is by default safe for parallel programming, and mainstream programming is by default dangerous.

Now, that's not to say that you can't make mainstream programming safer by being careful, and lots and lots of technology is devoted to doing just that. Either the programmer has to be careful, or is supported by the system in some way, but nevertheless you can move in the direction of allowing yourself to do parallel evaluation.

The direction that you move in is all about gaining more precise control about what side effects can take place. The reason I think functional programming languages have a lot to offer here is that they're already sitting at the other end of the spectrum. If you have a mainstream programming language and you're wanting to move in a purely functional direction, perhaps not all the way, you're going to learn a lot from what happens in the purely functional world.

I think there's a lot of fruitful opportunities for cross-fertilization. That's why I think Haskell is well placed for this multi-core stuff, as I think people are increasingly going to look to languages like Haskell and say 'oh, that's where we can get some good ideas at least,' whether or not it's the actual language or concrete syntax that they adopt.

All of that said however – it's not to say that purely functional programming means parallelism without tears. You can't just take a random functional program and feed it into a compiler and expect it to run in parallel. In fact it's jolly difficult! Twenty years ago we thought we could, but it turned out to be very hard to do that for completely different reasons to side effects: rather to do with granularity. It's very, very easy to get lots of very, very tiny parallel things to do, but then the overheads of doing all of those tiny little things overwhelm the benefits of going parallel.

I don't want to appear to claim that functional programmers have parallel programming wrapped up – far from it! It's just that I think there's a lot to offer and the world will be moving in that direction one way or another.

You obviously had some foresight twenty years ago ...

I don't think it was that we were that clairvoyant – it was simply about doing one thing well ...

So would you have done anything different in the development of Haskell if you had the chance?

That's a hard one. Of course we could have been cleverer, but even with retrospect, I'm not sure that I can see any major thing that I would have done differently.

And what's the most interesting program you've seen written with Haskell?

That's an interesting question. At first I was going to say GHC which is the compiler for Haskell. But I think the most interesting one, the one that really made me sit up and take notice, was Conal Elliot's Functional Reactive Animation, called FRAN. He wrote this paper that burst upon the scene [at ICFP 1997].

What it allowed you to do is to describe graphical animations, so things like a bouncing ball.

How do you make a ball bounce on the screen? One way to do it is to write a program that goes round a loop and every time it goes around the loop it figures out whether the ball should be one time step further on. It erases the old picture of the ball and draws a new picture. That's the way most graphics are done one way or another, but it's certainly hard to get right.

Another way to do it is, instead of repainting the screen, to say here is a value, and that value describes the position of the ball at any time. How can a value do that? Conal's said 'Just give me a function, and the value I'll produce will be a function from time to position. If I give you this function you can apply it at any old time and it will tell you where the ball is.' So all this business of repainting the screen can be re-delegated to another piece of code, that just says 'I'm ready to repaint now, so let me reapply this function and that will give me a picture and I'll draw that.'

So from a rather imperative notion of values that evolve over time, it turned it into a purely declarative idea of a value that describes the position of the ball at any time. Based on that simple idea Conal was able to describe lots of beautiful animations and ways of describing dynamics and things moving around and bouncing into one another in a very simple and beautiful way. And I had never thought of that. It expanded my idea of what a value might be.

What was surprising about it was that I didn't expect that that could be done in that way at all, in fact I had never thought about it. Haskell the language had allowed Conal to think sophisticated thoughts and express them as a programmer, and I thought that was pretty cool. This actually happens quite a lot as Haskell is a very high-level programming language, so people that think big thoughts can do big things in it.

What do you mean when you call a language 'lazy'?

Normally when you call a function, even in a call by value or strict functional programming language, you would evaluate the argument, and then you'd call the function. For example, once you called f on $3 + 4$, your first step would be to evaluate $3 + 4$ to make 7, then you'd call f and say you're passing it 7.

In a lazy language, you don't evaluate the $3 + 4$ because f might ignore it, in which case all that work computing $3 + 4$ was wasted. In a lazy language, you evaluate expressions only when their value is actually required, not when you call a function – it's call by need. A lazy person waits until their manager says 'I really need that report now,' whereas an eager will have it in their draw all done, but maybe their manager will never ask for it.

Lazy evaluation is about postponing tasks until you really have to do them. And that's the thing that distinguishes Haskell from ML, or Scheme for example.

If you're in a lazy language, it's much more difficult to predict the order of evaluation. Will this thing be evaluated at all, and if so, when, is a tricky question to answer. So that makes it much more difficult to do input/output. Basically, in a functional language, you shouldn't be doing input/output in an expression because input/output is a side effect.

In ML or Scheme, they say, 'oh well, we're functional most of the time, but for input/output we'll be non-functional and we'll let you do side effects and things that are allegedly functions.' They're not really functions however, as they have side effects. So you can call f and you can print something, or launch the missiles. In Haskell, if you call f, you can't launch the missiles as it's a function and it doesn't have any side effects.

In theory, lazy evaluation means that you can't take the ML or Scheme route of just saying 'oh well, we'll just allow you to do input/output side effects,' as you don't know what order they'll happen in. You wouldn't know if you armed the missiles before launching them, or launched them before arming them.

Because Haskell is lazy it meant that we were much more consistent about keeping the language pure. You could have a pure, strict, call by value language, but no one has managed to do that because the moment you have a strict call by value language, the temptation to add impurities (side effects) is overwhelming. So 'laziness kept us pure' is the slogan!

Do you know of any other pure languages?

Miranda, designed by David Turner, which has a whole bunch of predecessor languages, several designed by David Turner – they're all pure. Various subsets of Lisp are pure. But none widely

used ... oh, and CLEAN is pure(!). But for purely functional programming HASKELL must be the brand leader.

Do you think that lazy languages have lots of advantages over non-lazy languages?

I think probably on balance yes, as laziness has lots of advantages. But it has some disadvantages too, so I think the case is a bit more nuanced there [than in the case of purity].

A lazy language has ways of stating 'use call by value here,' and even if you were to say 'oh, the language should be call by value strict' (the opposite of lazy), you'd want ways to achieve laziness anyway. Any successor language [to HASKELL] will have support for both strict and lazy functions. So the question then is: what's the default, and how easy is it to get to these things? How do you mix them together? So it isn't kind of a completely either/or situation any more. But on balance yes, I'm definitely very happy with using the lazy approach, as that's what made HASKELL what it is and kept it pure.

You sound very proud of Haskell's purity

That's the thing. That's what makes HASKELL different. That's what it's about.

Do you think Haskell has been successful in creating a standard for functional programming languages?

Yes, again not standard as in the ISO standard sense, but standard as a kind of benchmark or brand leader for pure functional languages. It's definitely been successful in that. If someone asks, 'tell me the name of a pure functional programming language,' you'd say HASKELL. You could say CLEAN as well, but CLEAN is less widely used.

How do you respond to criticism of the language, such as this statement from Wikipedia: *'While Haskell has many advanced features not found in many other programming languages, some of these features have been criticized for making the language too complex or difficult to understand. In addition, there are complaints stemming from the purity of Haskell and its theoretical roots'*?

Partly it's a matter of taste. Things that one person may find difficult to understand, another might not. But also it's to do with doing one thing well again. HASKELL does take kind of an extreme approach: the language is pure, and it has a pretty sophisticated type system too. We've used HASKELL in effect as a laboratory for exploring advanced type system ideas. And that can make things complicated.

I think a good point is that HASKELL is a laboratory: it's a lab to explore ideas in. We intended it to be usable for real applications, and I think that it definitely is, and increasingly so. But it wasn't intended as a product for a company that wanted to sell a language to absolutely as many programmers as possible, in which you might take more care to make the syntax look like C, and you might think again about introducing complex features as you don't want to be confusing.

HASKELL was definitely designed with programmers in mind, but it wasn't designed for your average C++ programmer. It's to do not with smartness but with familiarity; there's a big mental rewiring process that happens when you switch from C++ or PERL to HASKELL. And that comes just from being a purely functional language, not because it's particularly complex. Any purely functional language requires you to make that switch.

If you're to be a purely functional programming language, you have to put up with that pain. Whether it's going to be the way of the future and everybody will do it – I don't know. But I think it's worth some of us exploring that. I feel quite unapologetic about saying that's what HASKELL is – if you don't want to learn purely functional programming or it doesn't feel comfortable to you or you don't want to go through the pain of learning it, well, that's a choice you can make. But it's worth being clear about what you're doing and trying to do it in a very clear and consistent and continuous way.

HASKELL, at least with GHC, has become very complicated. The language has evolved to become increasingly complicated as people suggest features, and we add them, and they have to interact with other features. At some point, maybe it will become just too complicated for any mortal to keep their head around, and then perhaps it's time for a new language – that's

the way that languages evolve.

Do you think that any language has hit that point yet, whether Haskell, C++ etc?

I don't know. C++ is also extremely complicated. But long lived languages that are extremely complicated also often have big bases of people who like them and are familiar with them and have lots of code written in them.

C++ isn't going to die any time soon. I don't think Haskell's going to die any time soon either, so I think there's a difficult job in balancing the complexity and saying 'well, we're not going to do any more, I declare that done now, because we don't want it to get any more complicated.' People with a big existing investment in it then ask 'oh, can you just do this,' and the 'just do this' is partly to be useful to them, and also because that's the way I do research.

I'm sitting in a lab and people are saying 'why don't you do that?,' and I say 'oh, that would be interesting to try so we find out.' But by the time we've logged all changes in it's very complicated, so I think there's definite truth in that Wikipedia criticism.

And on a side note, what attracted you to Microsoft research? How has the move affected your Haskell work?

I've been working in universities for about 17 years, and then I moved to Microsoft. I enjoyed working at universities a lot, but Microsoft was an opportunity to do something different. I think it's a good idea to have a change in your life every now and again. It was clearly going to be a change of content, but I enjoyed that change.

Microsoft has a very open attitude to research, and that's one of those things I got very clear before we moved. They hire good people and pretty much turn them loose. I don't get told what to do, so as far as my work on Haskell or GHC or research generally is concerned, the main change with moving to Microsoft was that I could do more of it, as I wasn't teaching or going to meetings etc. And of course all of those things were losses in a way and the teaching had it's own rewards.

Do you miss the teaching?

Well I don't wake up on Monday morning and wish I was giving a lecture! So I guess [I miss it] in theoretical way and not in a proximate kind of way. I still get to supervise graduate students.

Microsoft have stuck true to their word. I also get new opportunities [that were not available to me at university], as I can speak to developers inside the [Microsoft] firewall about functional programming in general, and Haskell in particular, which I never could before. Microsoft are completely open about allowing me to study what I like and publish what I like, so it's a very good research setup – it's the only research lab I know like that. It's fantastic – it's like being on sabbatical, only all the time.

Do you ever think the joke about Microsoft using Haskell as its standard language had come true? Haskell.NET?

Well, there are two answers to this one – the first would be of course, yes, that would be fantastic! I really think that functional programming has such a lot to offer the world.

As for the second, I don't know if you know this, but Haskell has a sort of unofficial slogan: avoid success at all costs. I think I mentioned this at a talk I gave about Haskell a few years back and it's become sort of a little saying. When you become too well known, or too widely used and too successful (and certainly being adopted by Microsoft means such a thing), suddenly you can't change anything anymore. You get caught and spend ages talking about things that have nothing to do with the research side of things.

I'm primarily a programming language researcher, so the fact that Haskell has up to now been used for just university types has been ideal. Now it's used a lot in industry but typically by people who are generally flexible, and they are generally a self selected rather bright group. What that means is that we could change the language and they wouldn't complain. Now, however, they're starting to complain if their libraries don't work, which means that we're beginning to get caught in the trap of being too successful.

What I'm really trying to say is that the fact Haskell hasn't become a real mainstream programming language, used by millions of developers, has allowed us to become much more

nimble, and from a research point of view, that's great. We have lots of users so we get lots of experience from them. What you want is to have a lot of users but not too many from a research point of view – hence the 'avoid success at all costs.'

Now, but at the other extreme, it would be fantastic to be really, really successful and adopted by Microsoft. In fact you may know my colleague down the corridor, Don Syme, who designed a language: F#. F# is somewhere between Haskell and C# – it's a Microsoft language, it's clearly functional but it's not pure and it's defining goal is to be a .NET language. It therefore takes on lots of benefits and also design choices that cannot be changed from .NET. I think that's a fantastic design point to be in and I'm absolutely delirious that Don's done that, and that he's been successfully turning it into a product – in some ways because it takes the heat off me, as now there is a functional language that is a Microsoft product!

So I'm free to research and do the moderate success thing. When you talk to Don [in a forthcoming interview in the *A-Z of Programming Languages* series], I think you will hear him say that he's got a lot of inspiration from Haskell. Some ideas have come from Haskell into F#, and ideas can migrate much more easily than concrete syntax and implementation and nitty-gritty design choices.

Haskell is used a lot for educational purposes. Are you happy with this, being a former lecturer, and why do you think this is?

Functional programming teaches you a different perspective on the whole enterprise of writing programs. I want every undergraduate to learn to write functional programs. Now if you're going to do that, you have to choose if you are going to teach Haskell or ML or Clean. My preference would be Haskell, but I think the most important thing is that you should teach purely functional programming in some shape or form as it makes you a better imperative programmer. Even if you're going to go back to C++, you'll write better C++ if you become familiar with functional programming.

Have you personally taught Haskell to many students?

No, I haven't actually! While I was at Glasgow I was exclusively engaged in first year teaching of Ada, because that was at the time in the first year language that Glasgow taught, and Glasgow took the attitude that each senior professor should teach first year students, as they're the ones that need to be turned on and treated best. That's the moment when you have the best chance of influencing them – are they even going to take a second year course?

Did you enjoy teaching Ada?

Yes, it was a lot of fun. It's all computer science and talking to 200 undergraduates about why computing is such fun is always exciting.

You've already touched on why you think all programmers should learn to write functional programs. Do you think functional programming should be taught at some stage in a programmer's career, or it should be the first thing they learn?

I don't know – I don't actually have a very strong opinion on that. I think there are a lot of related factors, such as what the students will put up with! I think student motivation is very important, so teaching students a language they have heard of as their first language has a powerful motivational factor.

On the other hand, since students come with such diverse experiences (some of them have done heaps of programming and some of them have done none) teaching them a language which all of them aren't familiar with can be a great leveler. So if I was in a university now I'd be arguing the case for teaching functional programming as a first year language, but I don't think it's a sort of unequivocal, 'only an idiot would think anything else' kind of thing!

Some say dealing with standard IO in Haskell doesn't feel as 'functional' as some would expect. What's your opinion?

Well it's not functional – IO is a side effect as we discussed. IO ensures the launching of the missiles: do it now and do it in this order. IO means that it needs to be done in a particular order, so you say do this and then do that and you are mutating the state of the world. It clearly is a side effect to launch missiles so there's no two ways about it.

If you have a purely functional program, then in principle, all it can do is take a value and deliver a value as its result. When Haskell was first born, all it would do is consume a character string and produce a character string. Then we thought, 'oh, that's not very cool, how can we launch missiles with that?' Then we thought, 'ah, maybe instead of a string, we could produce a list of commands that tell the outside world to launch the missiles and write to the disk.' So that could be the result value. We'd still produced a value – that was the list of commands, but somebody else was doing the side effects as it were, so we were still holy and pure!

Then the next challenge was to producing value that said read a file and to get the contents of the file into the program. But we wrote a way of doing that, but it always felt a bit unsatisfactory to me, and that pushed us to come up with the idea of monads. Monads provided the way we embody IO into Haskell; it's a very general idea that allows you to have a functional program that still includes side effects. I've been saying that purely functional programming means no effects, but programming with monads allows you to mix bits of program that do effect and bits that are pure without getting to two mixed up. So it allows you to not be one or the other.

But then, to answer your question, IO using monads still doesn't look like purely functional programming, and it shouldn't because it isn't. It's monadic programming, which is kept nicely separate and integrates beautifully with the functional part. So I suppose it's correct to say that it doesn't feel functional because it isn't, and shouldn't be.

What Haskell has given to the world, besides a laboratory to explore ideas in, is this monadic idea. We were stuck not being able to do IO well for quite a while. F# essentially has monads, even though it's an impure language, and so could do side effects. Nevertheless Don has imported into F# something he calls workflows, which are just a flimsy friendly name for monads. This is because even though F# is impure, monads are an idea that's useful in their own right. Necessity was the mother of invention.

So monads would be Haskell's lasting legacy in your eyes?

Yes, monads are a big deal. The idea that you can make something really practically useful for large scale applications out of a simple consistent idea is purely functional programming. I think that is a big thing that Haskell's done – sticking to our guns on that is the thing we're happiest about really.

One of the joys of being a researcher rather than somebody who's got to sell a product is that you can stick to your guns, and Microsoft have allowed me to do that.

What do you think Haskell's future will look like?

I don't know. My guess is that the language, as it is used by a lot of people, will continue to evolve in a gentle sort of way.

The main thing I'm working on right now is parallelism, multi-cores in particular, and I'm working with some colleagues in Australia at the University of NSW. I'm working very closely with them on something called nested data parallelism.

We've got various forms of parallelism in various forms of Haskell already, but I'm not content with any of them. I think that nested data parallelism is my best hope for being able to really employ tens or hundreds of processes rather than a handful. And nested data parallelism relies absolutely on being within a functional programming language. You simply couldn't do it in an imperative language.

And how far along that track are you? Are you having some success?

Yes, we are having some success. It's complicated to do and there's real research risk about it – we might not even succeed. But if you're sure you're going to succeed it's probably not research! We've been working on this for a couple of years. We should have prototypes that other people can work on within a few months, but it will be another couple of years before we know if it really works well or not.

I am quite hopeful about it – it's a pretty radical piece of compiler technology. It allows you to write programs in a way that's much easier for a programmer to write than conventional parallel programming. The compiler shakes the program about a great deal and produces a program that's easy for the computer to run. So it transforms from a program that's easy to write into a program that's easy to run. That's the way to think of it. The transformation is pretty radical

– there's a lot to do and if you don't do it right, the program will work but it will run much more slowly than it should, and the whole point is to go fast. I think it's [purely-functional programming] the only current chance to do this radical program transformation.

In the longer term, if you ask where Haskell is going, I think it will be in ideas, and ultimately in informing other language design. I don't know if Haskell will ever become mainstream like Java, C++ or C# are. I would be perfectly content if even the ideas in Haskell became mainstream. I think this is a more realistic goal – there are so many factors involved in widespread language adoption – ideas are ultimately more durable than implementations.

So what are you proudest of in terms of the languages development and use?

I think you can probably guess by now! Sticking to purity, the invention of monads and type classes. We haven't even talked about type classes yet. I think Haskell's types system, which started with an innovation called type classes, has proved extremely influential and successful. It's one distinctive achievement that was in Haskell since the very beginning. But even since then, Haskell has proved to be an excellent type system laboratory. Haskell has lots of type system features that no other language has. I'm still working on further development of this, and I'm pretty proud about that.

And where do you think computer languages will be heading in the next 5-20 years or so? Can you see any big trends etc?

It's back to effects. I don't know where programming in general will go, but I think that over the next 10 years, at that sort of timescale, we'll see mainstream programming becoming much more careful about effect – or side effects. That's my sole language trend that I'll forecast. And of course, even that's a guess, I'm crystal ball gazing.

Specifically, I think languages will grow pure or pure-ish subsets. There will be chunks of the language, even in the main imperative languages, that will be chunks that are pure.

Given all of your experience, what advice do you have for students or up and coming programmers?

Learn a wide range of programming languages, and in particular learn a functional language. Make sure that your education includes not just reading a book, but actually writing some functional programs, as it changes the way you think about the whole enterprise of programming. It's like if you can ski but you've never snowboarded: you hop on a snowboard and you fall off immediately. You initially think humans can't do this, but once you learn to snowboard it's a different way of doing the same thing. It's the same with programming languages, and that radically shifted perspective will make you a better programmer, no matter what style of programming you spend most of your time doing. It's no good just reading a book, you've got to write a purely functional program. It's not good reading a book about snow boarding – you have to do it and fall off a lot before you train your body to understand what's going on.

Thanks for taking the time to chat to me today. Is there anything else you'd like to add?

I'll add one other thing. Another distinctive feature of Haskell is that it has a very nice community. We haven't talked about the community at all, but Haskell has an extremely friendly sort of ecosystem growing up around it. There's a mailing list that people are extremely helpful on, it has a wiki that is maintained by the community and it has an IRC channel that hundreds of people are on being helpful. People often comment that it seems to be an unusually friendly place, compared to experiences they've had elsewhere (and I can't be specific about this as I genuinely don't know). I don't know how to attribute this, but I'm very pleased that the Haskell community has this reputation as being a friendly and welcoming place that's helpful too. It's an unusually healthy community and I really like that.

INTERCAL: Don Wood

In this interview, Computerworld ventures down a less serious path and chats to Don Woods about the development and uses of INTERCAL. Woods currently works at Google, following the company's recent acquisition of Postini, and he is best known for co-writing the original Adventure game with Will Crowther. He also co-authored The Hackers Dictionary. *Here we chat to him about all things spoof and the virtues of tonsils as removable organs*

How did you and James Lyon get the urge to create such an involved spoof language?

I'm not entirely sure. As indicated in the preface to the original reference manual, we came up with the idea (and most of the initial design) in the wee hours of the morning. We had just finished our – let's see, it would have been our freshman year – final exams and were more than a little giddy!

My recollection, though fuzzy after all these years, is that we and another friend had spent an earlier late-night bull session coming up with alternative names for spoken punctuation (spot, spark, spike, splat, wow, what, etc.) and that may have been a jumping off point in some way.

Why did you choose to spoof Fortran and COBOL in particular?

We didn't. (Even though Wikipedia seems to claim we did.) We spoofed the languages of the time, or at least the ones we were familiar with. (I've never actually learned COBOL myself, though I believe Jim Lyon knew the language.) The manual even lists the languages we were comparing ourselves to. And then we spoofed the reference manuals of the time, especially IBM documentation, again since that's what we were most familiar with. Admittedly, the language resembles FORTRAN more than it does, say, SNOBOL or APL, but then so do most computer languages.

What prompted the name Compiler Language With No Pronounceable Acronym? And how on earth did you get INTERCAL out of this?

I think we actually started with the name INTERCAL. I'm not sure where it came from; probably it just sounded good. (Sort of like FORTRAN is short for 'Formula Translation,' INTERCAL sounds like it should be short for something like 'Interblah Calculation'). I don't remember any more specific etymology. Then when we wanted to come up with an acronym, one of us thought of the paradoxical 'Compiler Language With No Pronounceable Acronym.'

How long did it take to develop INTERCAL? Did you come across any unforeseen problems during the initial development period?

That depends on what you mean by 'develop.' We designed the language without too much trouble. Writing the manual took a while, especially for things like the circuit diagrams we included as nonsensical illustrations. The compiler itself actually wasn't too much trouble, given that we weren't at all concerned with optimising the performance of either the compiler or the compiled code.

Our compiler converted the INTERCAL program to SNOBOL (actually SPITBOL, which is a compilable version of SNOBOL) and represented INTERCAL datatypes using character strings in which all the characters were 0s and 1s.

Do you use either C-INTERCAL or CLC-INTERCAL currently?

No, though I follow the *alt.lang.intercal* newsgroup and occasionally post there.

Have you ever actually tried to write anything useful in INTERCAL that actually works? Has anyone else?

Me, no. Others have done so. I remember seeing a Web page that used INTERCAL (with some I/O extensions no doubt) to play the game 'Bugs and Loops,' in which players add rules to a Turing machine trying to make the machine run as long as possible without going off the end of its tape or going into an infinite loop.

How do you feel given that the language was created in 1972, and variations of it

are still being maintained? Do you feel like you have your own dedicated following of spoof programmers now?

I admit I'm surprised at its longevity. Some of the jokes in the original work feel rather dated at this point. It helps that the language provides a place where people can discuss oddball features missing from other languages, such as the 'COME FROM' statement and operators that work in base 3.

And no, I don't feel like a have a 'following,' though every once in a while I do get caught off-guard by someone turning out to be an enthusiastic INTERCAL geek. When I joined Google some months back, someone apparently noticed my arrival and took the opportunity to propose adding a style guide for INTERCAL to go alongside Google's guides for C++, JAVA and other languages. (The proposal got shot down, but the proposed style guide is still available internally.)

Did you have a particular person in mind when you wrote the following statement in the reference manual: 'It is a well-known and oft-demonstrated fact that a person whose work is incomprehensible is held in high esteem'?

Oddly, I don't think we had anyone specific in mind.

Do you know of anyone who has been promoted because they demonstrated their superior technical knowledge by showing off an INTERCAL program?

Heh, no.

The footnotes of the manual state: '4) Since all other reference manuals have Appendices, it was decided that the INTERCAL manual should contain some other type of removable organ.' We understand why you'd want to remove the appendix, no one likes them and they serve no purpose, but tonsils seem to be much more useful. Do you regret your decision to pick the tonsil as the only removable organ?

No, I never gave that much thought. We were pleased to have come up with a second removable organ so that we could make the point of not including an appendix. Besides, just because it's removable doesn't mean it's not useful to have it!

Did you struggle to make INTERCAL Turing-complete?

Struggle? No. We did want to make sure the language was complete, but it wasn't all that hard to show that it was.

How do you respond to criticism of the language, such as this point from Wikipedia: '*A Sieve of Eratosthenes benchmark, computing all prime numbers less than 65536, was tested on a Sun SPARCStation-1. In C, it took less than half a second; the same program in INTERCAL took over seventeen hours*'?

Excuse me? That's not criticism, that's a boast! Back in our original implementation on a high-end IBM mainframe (IBM 360/91), I would boast that a single 16-bit integer division took 30 seconds, and claimed it as a record!

Would you do anything differently if you had the chance to develop INTERCAL again now?

I'm sure there are fine points I'd change, and I'd include some of the more creative features that others have proposed (and sometimes implemented) over the years. Also, some of the jokes and/or language features are a bit dated now, such as the XOR operator being a 'V' overstruck with a '−,' and our mention of what this turns out to be if the characters are overstruck on a punched card.

In your opinion, has INTERCAL contributed anything useful at all to computer development?

Does entertainment count? :-) I suppose there are also second-order effects such as giving some people (including Lyon and myself) a chance to learn about compilers and the like.

Perhaps more important, when you have to solve problems without using any of the usual tools, you can sometimes learn new things. In 2003 I received a note from Knuth saying he had 'just spent a week writing an INTERCAL program' that he was posting to his 'news' Web page,

and while working on it he'd noticed that 'the division routine of the standard INTERCAL library has a really cool hack that I hadn't seen before.' He wanted to know if I could remember which of Lyon or myself had come up with it so he could give proper credit when he mentioned the trick in volume 4 of The Art of Computer Programming. (I couldn't recall.)

Where do you envisage INTERCAL's future lying?

I've no idea, seeing as how I didn't envisage it getting this far!

Has anyone ever accidentally taken INTERCAL to be a serious programming language?

Heavens, I hope not! (Though I was concerned YOU had done so when you first contacted me!)

Have you been impressed by any other programming languages, such as Brain**?**

I've looked at a few other such languages but never spent a lot of time on them. Frankly, the ones that impress me more are the non-spoof languages that have amazingly powerful features (usually within limited domains), such as APL's multidimensional operations or SNOBOL's pattern matching. (I'd be curious to go back and look at SNOBOL again now that there are other languages with powerful regular-expression operators.)

The closest I've come to being impressed by another 'limited' programming language was a hypothetical computer described to me long ago by a co-worker who was a part-time professor at Northeastern University. The computer's memory was 65536 bits, individually addressable using 16-bit addresses. The computer had only one type of instruction; it consisted of 48 consecutive bits starting anywhere in memory. The instruction was interpreted as three 16-bit addresses, X Y Z, and the operation was 'copy the bit from location X to location Y, then to execute the instruction starting at location Z.' The students were first tasked with constructing a conditional branch (if bit A is set go to B, else go to C). I think the next assignment was to build a 16-bit adder. Now THAT's minimalist!

Where do you see computer programming languages heading in the near future?

An interesting question, but frankly it's not my field so I haven't spent any time pondering the matter. I do expect we'll continue to see a growing dichotomy between general programming languages (Perl, Python, C++, Java, whatever) and application-level languages (suites for building Web-based tools and such). It seems that we currently have people who use the general programming languages, but don't have any understanding of what's going on down at the microcode or hardware levels.

Do you have any advice for up-and-coming spoof programmers?

Try to find a niche that isn't already filled. Hm, you know, SPOOF would be a fine name for a language. It's even got OO in the name!

And finally, as already discussed with Bjarne Stroustrup, do you think that facial hair is related to the success of programming languages?

I hadn't seen that theory before, but it's quite amusing.

I don't think I had any facial hair when we designed INTERCAL, but I've been acquiring more over the years. Maybe that's why INTERCAL's still thriving?

Is there anything else you'd like to add?

In some sense INTERCAL is the ultimate language for hackers, where I use 'hacker' in the older, non-criminal sense, meaning someone who enjoys figuring out how to accomplish something despite the limitations of the available tools. (One of the definitions in *The Hacker's Dictionary* is 'One who builds furniture using an axe.') Much of the fun of INTERCAL comes from figuring out how it can be used to do something that would be trivial in other languages. More fun is had by extending the language with weird new features and then figuring out what can be done by creative use of those features.

JavaScript: Brendan Eich

Brendan Eich is the creator of JavaScript *and Chief Technology Officer of Mozilla Corporation. Eich details the development of JS from its inception at Netscape in 1995, and comments on its continued popularity, as well as what he believes will be the future of client-side scripting languages on the Web*

What prompted the development of JavaScript?

I've written about the early history on my blog:
`http://Weblogs.mozillazine.org/roadmap/archives/2008/04/popularity.html`.

I joined Netscape on 4 April 1995, with the goal of embedding the Scheme programming language, or something like it, into Netscape's browser. But due to requisition scarcity, I was hired into the Netscape Server group, which was responsible for the Web server and proxy products. I worked for a month on next-generation HTTP design, but by May I switched back to the group I'd been recruited to join, the Client (browser) team, and I immediately started prototyping what became JavaScript.

The impetus was the belief on the part of at least Marc Andreessen and myself, along with Bill Joy of Sun, that HTML needed a 'scripting language,' a programming language that was easy to use by amateurs and novices, where the code could be written directly in source form as part of the Web page markup. We aimed to provide a 'glue language' for the Web designers and part-time programmers who were building Web content from components such as images, plugins, and Java applets. We saw Java as the 'component language' used by higher-priced programmers, where the glue programmers – the Web page designers – would assemble components and automate their interactions using JS.

In this sense, JS was analogous to Visual Basic, and Java to C++, in Microsoft's programming language family used on Windows and in its applications. This division of labor across the programming pyramid fosters greater innovation than alternatives that require all programmers to use the 'real' programming language (Java or C++) instead of the 'little' scripting language.

So was there a particular problem you were trying to solve?

The lack of programmability of Web pages made them static, text-heavy, with at best images in tables or floating on the right or left. With a scripting language like JS that could touch elements of the page, change their properties, and respond to events, we envisioned a much livelier Web consisting of pages that acted more like applications.

Indeed, some early adopters, even in late 1995 (Netscape 2's beta period), built advanced Web apps using JS and frames in framesets, prefiguring the Ajax or Web 2.0 style of development. But machines were slower then, JS had a relatively impoverished initial set of browser APIs, and the means to communicate with servers generally involved reloading whole Web pages.

How did JavaScript get its name given that it's essentially unrelated to the Java programming language?

See my blog post, linked above.

Why was JS originally named Mocha and then LiveScript?

Mocha was Marc Andreessen's code name, but Netscape marketing saw potential trademark conflicts and did not prefer it on other grounds. They had a 'live' meme going in their naming (LiveWire, LiveScript, etc). But the Java momentum of the time (1995-1996) swept these before it.

How does JavaScript differ from ECMAScript?

ECMA-262 Edition 3 is the latest ECMAScript standard. Edition 1 was based on my work at Netscape, combined with Microsoft's reverse-engineering of it (called JScript) in IE, along with a few other workalikes from Borland and a few other companies.

The 3rd edition explicitly allows many kinds of extensions in its Chapter 16, and so JavaScript means more than just what is in the standard, and the language is evolving ahead of the standard in implementations such as Mozilla's SpiderMonkey and Rhino engines (SpiderMonkey is

the JS engine in Firefox).

The ECMA standard codifies just the core language, not the DOM, and many people think of the DOM as 'JavaScript.'

Do you believe that the terms JavaScript and JScript can or should be used interchangeably?

JScript is not used, much or at all in cross-browser documentation and books, to refer to the language. JavaScript (JS for short) is what all the books use in their titles, what all the developer docs and conferences use, etc. It's the true name, for better and worse.

Were there any particularly hard/annoying problems you had to overcome in the development of the language?

Yes, mainly the incredibly short development cycle to prove the concept, after which the language design was frozen by necessity. I spent about ten days in May 1995 developing the interpreter, including the built-in objects except for the Date class (Ken Smith of Netscape helped write that by translating Java's `java.util.Date` class to C, unintentionally inheriting `java.util.Date`'s Y2K bugs in the process!)

I spent the rest of 1995 embedding this engine in the Netscape browser and creating what has become known as the DOM (Document Object Model), specifically the DOM level 0: APIs from JS to control windows, documents, forms, links, images, etc., and to respond to events and run code from timers.

I was the lone JS developer at Netscape until mid-1996.

What is the most interesting program that you've seen written with JavaScript?

TIBET (`http://www.technicalpursuit.com`) was an early, ambitious framework modeled on Smalltalk.

There are amazing things in JS nowadays, including HotRuby[3] – this runs Ruby bytecode entirely in JS in the browser – and a Java VM[4].

We are seeing more games, both new and ported from other implementations as well:
`http://blog.nihilogic.dk/2008/04/super-mario-in-14kb-javascript.html`
`http://canvex.lazyilluminati.com/83/play.xhtml`.

And John Resig's port of the Processing visualization language takes the cake:
`http://ejohn.org/blog/processingjs`.

And what's the worst?

I couldn't possibly pick one single worst JS program. I'll simply say that in the old days, JS was mainly used for annoyances such as pop-up windows, status bar scrolling text, etc. Good thing browsers such as Firefox evolved user controls, with sane defaults, for these pests. Netscape should have had such options in the first place.

Have you ever seen the language used in a way that was not originally intended? If so, what was it? And did it or didn't it work?

The Java VM (Orto) mentioned above is one example. I did not intend JS to be a 'target' language for compilers such as Google Web Toolkit (GWT) or (before GWT) HaXe and similar such code generators, which take a different source language and produce JS as the 'object' or 'target' executable language.

The code generator approach uses JS as a safe mid-level intermediate language between a high-level source language written on the server side, and the optimized C or C++ code in the browser that implements JS. This stresses different performance paths in the JS engine code, and potentially causes people to push for features in the ECMA standard that are not appropriate for most human coders.

JS code generation by compilers and runtimes that use a different source language does seem to be working, in the sense that JS performance is good enough and getting better, and everyone

[3] `http://ejohn.org/blog/ruby-vm-in-javascript`

[4] Orto, see `http://ejohn.org/blog/running-java-in-javascript` but beware: I'm not sure how much of the Java VM is implemented in JS – still, it's by all accounts an impressive feat

wants to maximize 'reach' by targeting JS in the browser. But most JS is hand-coded, and I expect it will remain so for a long time.

It seems that many cross-site scripting exploits involve JavaScript. How do you feel about this? Are there plans to solve some of these problems?

Yes, we have plans to address these, both through the standards bodies including the W3C, and through content restrictions that Web developers can impose at a fine grain. See the document `http://www.gerv.net/security/content-restrictions` and the Mozilla bug tracking work to implement these restrictions: `https://bugzilla.mozilla.org/show_bug.cgi?id=390910`.

When do you expect the next version of JavaScript to be released? Do you have in mind any improvements that will be incorporated?

I expect the 3.1 edition of the ECMA-262 standard will be done by the middle of 2009, and I hope that a harmonized 4th edition will follow within a year. It's more important to me (and I believe to almost everyone on the committee) that new editions of the specification be proven by multiple interoperating prototype implementations, than the specs be rushed to de-jure approval by a certain date. But the 3.1 effort seems achievable in the near term, and a harmonized major 4th edition should be achievable as a compatible successor in a year or two.

The improvements in the 3.1 effort focus on bug fixes, de-facto standards developed in engines such as SpiderMonkey (e. g. getters and setters) and reverse-engineered in other browsers, and affordances for defining objects and properties with greater integrity (objects that can't be extended, properties that can't be overwritten, etc.).

The improvements for the harmonized major edition following 3.1 simply build on the 3.1 additions and focus on usability (including new syntax), modularity, further integrity features, and in general, solutions to programming-in-the-large problems in the current language.

How do you feel about the place of JavaScript in Web 2.0?

It's clear JS was essential to the Ajax or Web 2.0 revolution. I would say Firefox, Safari, and renewed browser competition, and the renewed Web standards activities they spawned, were also important.

Real programs run in browsers too, and they are written in JS. But JS had to be sufficiently capable as a precondition for all of this progress to occur, even in the older Internet Explorer browser versions (IE 5.5, IE 6), which were barely maintained by Microsoft for the first five years of the new millennium. So JS was the tap root.

How do you feel about all the negative vibes expressed towards JavaScript over the years?

These vibes seem to me to be a mix of:

- Early objections to the idea of a scripting language embedded in HTML.
- Appropriate rejection of the annoyance features JS enabled (and lack of sane controls, e. g. over pop-ups, until browsers such as Firefox came along).
- Confusion of DOM incompatibilities among browsers, which caused developer pain, with the generally more compatible JS implementations, which caused much less (but non-zero) pain.
- And of course, some people still feel negatively about the Netscape marketing scam of naming the language JavaScript, implying a connection with Java, if not intentionally sowing confusion between JS and Java (for the record, I don't believe anyone at Netscape intended to sow such confusion).

These negative vibes are understandable. JS is the only example of a programming language that must interoperate at Web scale (wider than any other platform), on multiple operating systems and in many competing browsers. Other programming languages supported by browser plugins come from single vendors, who can control interoperation better by single-sourcing the implementation. Therefore JS and the DOM it controls have been a rough interoperation ride for Web developers.

It did not help that Netscape and Microsoft fought a browser war that forced premature standardization after a furious period of innovation, and which ended with way too many years

of neglect of JS and other Web standards under the IE monopoly.

On the up side, many developers profess to like programming in JS, and it has experienced a true renaissance since 2004 and the advent of Web 2.0 or Ajax programming.

What do you think the future impact of JavaScript and other client-side scripting languages will be on the Web?

I think JavaScript will be the default, and only obligatory, programming language in browsers for a while yet. But other languages will be supported, at first in one or another browser, eventually in cross-browser standard forms. Mozilla's browsers, including Firefox, optionally support C-Python integration, but you have to build it yourself and make sure your users have the C-Python runtime. We are working on better ways to support popular languages safely, compatibly, and with automated download of up-to-date runtime code.

It's clear the client side of the Web standards deserves programmability, as Marc Andreessen and I envisioned in 1995. The desktop and mobile computers of the world have plenty of cycles and storage to do useful tasks (more now than ever), without having to restrict their automation capabilities to submitting forms or sending messages to real programs running on Web servers. Real programs run in browsers too, and they are written in JS.

The impact of JS is only increasing, as it becomes the standard for scripting not only in the browser, but on the desktop and in devices such as the iPhone.

How do you feel about the recent release of JavaScript frameworks like SproutCore and Objective-J/Cappuccino? What impact do you think these will have on the future of Web applications?

The Apple hype machine has certainly made some folks treat these as the second coming of Ajax. To me they are in a continuum of evolving JS libraries and frameworks, including Google GWT and such popular libraries as Dojo, JQuery, YUI, and Prototype. I don't particularly expect any one winner to take all, at least not for years, and then only in parts of the Web. On certain devices, of course, you may have essentially no choice, but the Web is wider than any one device, however popular.

Do you think that we are likely to see the death of desktop applications?

No, but I think you will see more desktop applications written using Web technologies, even if they are not hosted in a Web server. And of course Web apps will continue to proliferate. With the evolution of JS and other browser-based Web standards, we'll see Web apps capable of more interactions and performance feats that formerly could be done only by desktop apps. We are already seeing this with offline support, canvas 2D and 3D rendering, etc. in the latest generation of browsers.

How do you see the increasing popularity of plugins like Flash affecting the popularity of JavaScript?

Flash is doing its part to be a good Ajax citizen, to be scriptable from JS and addressable using URLs – to be a component on the page along with other components, whether plugins, built-in objects such as images or tables, or purely JS objects. The open Web levels everything upward, and militates against single-vendor lock-in. You can see this in how Flash has evolved to play well in the Web 2.0 world, and Microsoft's Silverlight also aims to integrate well into the modern Web-standards world.

People fear a return to proprietary, single-vendor plugins controlling the entire Web page and user experience, but I doubt that will happen all over the Web.

First, Web standards in the cutting edge browsers are evolving to compete with Flash and Silverlight on video, animation, high-performance JS, and so on.

Second, no Web site will sacrifice 'reach' for 'bling,' and plugins always lack reach compared to natively-implemented browser Web standards such as JS. Users do not always update their plugins, and users reject plugins while continuing to trust and use browsers.

Where do you envisage JavaScript's future lying?

Certainly in the browser, but also beyond it, in servers and as an end-to-end programming language (as well as in more conventional desktop or operating system scripting roles).

Do you still think that (as you once said): 'ECMAScript was always an unwanted trade name that sounds like a skin disease'?

I don't think about this much, but sure: it's not a desired name and it does sound a bit like eczema.

Do you still expect ECMA-262 to be ready by October 2008? Do you expect the new version to be backwards incompatible at all?

If you mean the 4th Edition of ECMA-262, no: we do not expect that in 2008, and right now the technical committee responsible (ECMA TC39) is working together to harmonize proposals for both a near-term (Spring 2009) 3.1 edition of ECMAScript, and a more expansive (but not too big) follow-on edition, which we've been calling the 4th edition.

Has the evolution and popularity of JS surprised you in anyway?

The popularity has surprised me. I was resigned for a long time to JS being unpopular due to those annoying popups, but more: due to its unconventional combination of functional and prototype-based object programming traditions. But it turns out that programmers, some who started programming with JS, others seasoned in the functional and dynamic OOP languages of the past, actually like this unconventional mix.

What are you proudest of in JavasScript's initial development and continuing use?

The combination of first-class functions and object prototypes. I would not say it's perfect, especially as standardized (mistakes were added, as well as amplified, by standardization). But apart from the glitches and artifacts of rushing, the keystone concepts hang together pretty well after all these years.

Where do you see computer programming languages heading in the future, particularly in the next 5 to 20 years?

There are two big problems facing all of us which require better programming languages:

- Multicore/massively-parallel computers that are upon us even now, on the desktop, and coming soon to mobile devices. Computer scientists are scrambling to make up for the lack of progress in the last 15 years making parallel computing easier and more usable. JS has its role to play in addressing the multi-core world, starting with relatively simple extensions such as Google Gears' worker pools – shared-nothing background threads with which browser JS communicates by sending and receiving messages.
- Security. A programming language cannot create or guarantee security by itself, since security is a set of end-to-end or system properties, covering all levels of abstraction, including above and below the language. But a programming language can certainly give its users better or worse tools for building secure systems and proving facts about those security properties that can be expressed in the language.

Do you have any advice for up-and-coming programmers?

Study the classics: Knuth, Wirth, Hoare. Computer science is a wheel, which rotates every 10-20 years in terms of academic research focus. Much that was discovered in the early days is still relevant. Of course great work has been done more recently, but from what I can tell, students get more exposure to the recent stuff, and almost none to the giants of the past.

Is there anything else you'd like to add?

Not now, I'm out of time and have to get back to work!

Lua: Roberto Ierusalimschy

We chat to Prof. Roberto Ierusalimschy about the design and development of Lua. Prof. Ierusalimschy is currently an Associate Professor in the Pontifical Catholic University of Rio de Janeiro's Informatics Department where he undertakes research on programming languages, with particular focus on scripting and domain specific languages. Prof. Ierusalimschy is currently supported by the Brazilian Council for the Development of Research and Technology as an independent researcher, and has a grant from Microsoft Research for the development of Lua.NET. He also has a grant from FINEP for the development of libraries for Lua.

What prompted the development of Lua? Was there a particular problem you were trying to solve?

In our paper for the Third ACM History of Programming Languages Conference we outline the whole story about the origins of Lua.

To make a long story short, yes, we did develop Lua to solve a particular problem. Although we developed Lua in an academic institution, Lua was never an 'academic language,' that is, a language to write papers about. We needed an easy-to-use configuration language, and the only configuration language available at that time (1993) was Tcl. Our users did not consider Tcl an easy-to-use language. So we created our own configuration language.

How did the name Lua come about?

Before Lua I had created a language that I called SOL, which stood for 'Simple Object Language' but also means 'sun' in Portuguese. That language was replaced by Lua (still nameless at that time). As we perceived Lua to be 'smaller' than Sol, a friend suggested this name, which means 'moon' in Portuguese.

Were there any particularly difficult problems you had to overcome in the development of the language?

No. The first implementation was really simple, and it solved the problems at hand. Since then, we have had the luxury of avoiding hard/annoying problems. That is, there have been many problems along the way, but we never had to overcome them; we have always had the option to postpone a solution.

Some of them have waited several years before being solved. For instance, since Lua 2.2, released in 1995, we have wanted lexical scoping in Lua, but we didn't know how to implement it efficiently within Lua's constraints. Nobody did. Only with Lua 5.0, released in 2003 did we solve the problem, with a novel algorithm.

What is the most interesting program that you've seen written with Lua and why?

I have seen many interesting programs written in Lua, in many different ways. I think it would be unfair to single one out. As a category, I particularly like table-driven programs, that is, programs that are more generic than the particular problem at hand and that are configured for that particular problem via tables.

Have you ever seen the language used in a way that was not originally intended? If so, what was it, and did it work?

For me, one of the most unexpected uses of Lua is `inline::Lua`, a Perl extension for embedding Lua scripts into Perl code. I always thought that it was a weird thing to use Lua for scripting a scripting language. It does work, but I do not know how useful it really is.

In a broader sense, the whole use of Lua in games was unexpected for us. We did not create Lua for games, and we had never thought about this possibility before. Of course, with hindsight it looks an obvious application area, and we are very happy to be part of this community. And it seems to be working ;)

You've mentioned the usage of Lua in games already and it's used for scripting in some very famous games such as World of Warcraft (WoW). Have you played WoW

or written scripts in it?

No :) I have never played that kind of games (RPG). Actually, until recently I had no knowledge at all about WoW add ons (what they call their scripts). In the last Lua Workshop, Jim Whitehead gave a nice talk about WoW add ons; it was only then that I learned the little I currently know about them.

Do you think that the use of Lua in computer games has allowed people to discover the language who may not have done so otherwise?

Sure. I guess more people have learned about Lua through games than through any other channel.

Do you think that computer games have a part to play in promoting programming languages?

Certainly games are an important way to introduce people to programming. Many kids start using computers to play games, so it seems natural to use this same stimulus for more advanced uses of computers and for programming. However, we should always consider other routes to stimulate people to learn programming. Not everybody is interested in games. In particular, girls are much less motivated by games (or at least by most games) than boys.

Over 40 percent of Adobe Lightroom is believed to be written in Lua. How do you feel about this?

Proud :)

Why do you think that Lua has been such a popular toolkit for programs like Adobe lightroom and various computer games?

There is an important difference between Adobe Lightroom and games in general. For most games, I think the main reason for choosing Lua is its emphasis on scripting. Lightroom has made a different use of Lua, as a large part of the program is written in Lua. For Adobe, a strong reason for choosing Lua was its simplicity. In all cases, however, the easiness of interfacing with C/C++ is quite important, too.

In the Wikipedia article on Lua, it notes: '*... Lua's creators also state that Lisp and Scheme with their single, ubiquitous data structure mechanism were a major influence on their decision to develop the table as the primary data structure of Lua.*' Is this true, and why was the list such a powerful influence?

Scheme has been a major source of inspiration for us. This is a language I would love to have created. And it is amazing what we can do using only lists. However, lists do not seem so appropriate for a language where the main paradigm is imperative, such as Lua. Associative arrays have proved to be quite a flexible mechanism.

Do you think that the MIT license has allowed the language to grow in popularity?

Sure. In the first year that we released Lua outside PUC, we adopted a more restricted license. Basically it was free for academic use but not for commercial use. It was only after we changed for a more liberal license that Lua started to spread. I guess that even a GPL-like license would hurt its spread. Most game companies are very secretive about their technologies. Sometimes, it is hard to know who is actually using Lua!

Do you think Lua has any significant flaws?

It is difficult for us to point to any clear flaw, otherwise we would have corrected it. But, like with any other language, the design of Lua involves many compromises. For instance, several people complain that its syntax is too verbose, but that syntax is friendlier to non programmers (such as gamers). So, for some people the syntax is flawed, for others it is not. Similarly, for some programmers even the dynamic typing is a flaw.

How does or will 5.1.3 differ from previous versions of Lua?

In Lua, 5.1.x are only bug-fix releases. So, 5.1.3 differs from 5.1 only in fixing the few bugs found in 5.1.2. The next 'real' release, 5.2, is still somewhat far on the horizon. Lua evolved somewhat quickly until version 5 (some users would say too quickly), so now we would like to

allow some time for its culture to stabilize. After all, each new version automatically outdates current books, extra documentation, and the like. So, we are currently not planning big changes for the language in the near future.

Has corporate sponsorship of the language influenced the way Lua has developed in any way?

The corporate sponsorship program is still very recent; it started in June 2008. But it has no influence whatsoever in the development of Lua. The program offers only visibility to the sponsor. Sponsors do not have any exclusive channel to express their wishes about the language, and we do not feel obliged in any way to accept their suggestions.

What impact do you feel the growth of open source has had on Lua?

A huge impact! The development of Lua does not follow the typical development structure of open source projects however; apart from this Lua is a typical product of the open source era. We have a strong community and we get lots of feedback from this community. Lua would never achieve its popularity and its quality if it was not open source.

Why do you think Lua is a popular choice for providing a scripting interface within larger applications?

Except for Tcl, Lua is the only language designed since day one for that specific purpose. As I said before, any language design generally has lots of compromises. Lua's compromises are directed at being good for scripting, that is, for controlling applications. Most other languages have different compromises, such as having more complete libraries, better integration with the operating system, or a more rigid object system.

One place where Lua seems to be used widely is in sysadmin tools such as Snort. What impact do you think sysadmins have on a language?

On some languages sysadmins may have a big impact. Perl, for instance, got very strongly influence from that area. But Lua has had very little impact from sysadmins. That's because their usage and their goals are quite different.

For instance, consider an application like *Snort* or *Wireshark*. The goal of Perl is to allow you to implement the entire application in Perl. For that, the language must provide all system primitives that those tools may ever need. Lua, on the other hand, emphasizes multi-language development. The primitives specific for the application are provided by the application itself, not by Lua.

Also, sysadmin support frequently conflicts with portability – a main goal in Lua. Again, a sysadmin tool should provide access to all facilities of the system, no matter how idiosyncratic they are. Lua has some libraries to allow such access, but they are not built-in. And even those libraries try to present system facilities in a more standard, less idiosyncratic way.

What languages do you currently work with?

The language I work with most nowadays is C, both in the implementation of Lua and in some libraries. I also use Lua frequently, for tasks such as text processing and system automation. In the past I have worked with several different languages: I have substantial programming with Fortran, Mumps, Snobol, Smalltalk, Scheme, Pascal and C++, plus assemblers for various machines.

Is there a particular tool which you feel could really do with having Lua embedded in it?

It is hard to think about a tool that would not benefit from an embedded scripting facility, and Lua is an obvious choice for that support.

In your opinion, what lasting legacy has Lua brought to computer development?

I think it is far too early to talk about any 'lasting' legacy from Lua. But I think Lua has had already some impact on language design. The notion of co-routines, as implemented in Lua, has brought some novelties to that area. Also the object model adopted by Lua, based in delegation, is often cited. In the implementation aspect, Lua was a big showcase for register-based virtual machines.

Lua is also a showcase for the idea that 'small is beautiful,' that software does not need to be bloated to be useful.

Where do you envisage Lua's future lying?

Scripting. It is a pity that the term 'scripting language' is becoming a synonym for 'dynamic language.' A scripting language, as its name implies, is a language that is mainly used for scripting. The origins of the name are the shell languages that have been used to script other programs. Tcl enlarged it for scripting a program, but later people started applying the term for languages like Perl or Python, which are not scripting languages (in that original meaning) at all. They are dynamic languages. For real scripting, Lua is becoming a dominant language.

What are you most proud of in terms of the language's initial development and continuing use?

I am very proud that Lua achieved all this popularity given where it came from. From all languages ever to achieve some level of popularity, Lua is the only one not created in a developed country. Actually, besides Lua and Ruby, I guess all those languages were created in the US or Western Europe.

Where do you see computer programming languages heading in the next 5 to 20 years?

The easy part of predicting the next 20 years is that it will take a long time to be proved wrong. But we may try the reverse: where were we 20 years back, in the 80's?

I am old enough to remember the Fifth Generation project. Many people claimed at that time that in the far future (which is now) we would all be programming in Prolog :) In the short term, Ada seemed set to become the dominant language in most areas. It is interesting that the seeds of relevant changes in programming languages were already there. Object-oriented programming was on the rise; OOPSLA was created in 1986. But at that time no one would have bet on C++ overcoming Ada.

So, I would say that the seeds for the next 20 years are already out there, but they are probably not what people think or expect.

Do you have any advice for up-and-coming programmers?

Learn Lua :)

More seriously, I really subscribe to the idea that 'if the only tool you have is a hammer, you treat everything like a nail.' So, programmers should learn several languages and learn how to use the strengths of each one effectively. It is no use to learn several languages if you do not respect their differences.

Modula-3: Luca Cardelli

Luca Cardelli is a member of the Modula-*3 design committee. Cardelli is a Principal Researcher and Head of the Programming Principles and Tools and Security groups at Microsoft Research in Cambridge, UK, and is an ACM Fellow. Here he chats to Computerworld about the origins of* Modula-*3, including how the most exciting* Modula-*3 design meeting ever was abruptly interrupted by the San Francisco 7.1 earthquake*

Why did you feel the need to develop Modula-3? Was it a reaction to a problem that needed solving?

The problem was developing programming environments in a type-safe language. This meant that if I wrote a type-safe library, and my clients had a hard crash, I could say: 'not my problem, somebody must be cheating somewhere' because the typechecker guaranteed that it wasn't my problem. You couldn't say that if you used C++.

Why was the name Modula-3 chosen?

We wanted to show continuity of the basic philosophy of modularization of Modula-2, carried out into an object-oriented language. Klaus Wirth designed Modula-2 while (or shortly after) visiting Xerox PARC, so there was a common origin. We asked him to use the name Modula-3, and he agreed, and he also occasionally attended our meetings.

How did Modula-2+ influence the design of Modula-3?

It was basically the same language, but with none of the dark corners.

Modula-2+ had been developing organically, and needed a cleanup and standardization. We also wanted to publicize the innovative features of Modula-2+ (which largely came from Cedar/Mesa at Xerox PARC), and make them available to a wider community.

Were there any particularly hard/annoying problems you had to overcome in the development of the language?

Settling the type system was the hard part, not only for me, but I believe for everybody. A POPL paper discussed just that part.

Why was one of the language's aims to continue the tradition of type safety, while introducing new elements for practical real-world programming? Was there a real need for this in the 1980s?

Yes, the idea to design type-safe operating systems was still in full swing. It started at Xerox with Cedar/Mesa, and continued at DEC with the Taos operating system. You might say it is still continuing with Microsoft's .NET, and we are not quite there yet.

What is the most interesting program that you've seen written with Modula-3?

I'll just talk about my programs. I wrote the second program (after Next Computer's) direct-manipulation user interface editor. And I wrote the Obliq distributed programming language, which was heavily based on Modula-3's network objects.

Have you ever seen the language used in a way that was not originally intended? If so, what was it? And did it or didn't it work?

Not really; we intended to support type-safe systems programming and that is what happened. It's possible that we missed some opportunities, however.

Why do you think that the language hasn't been widely adopted by industry, but is still influential in research circles?

Basically, competition from Java. Java had all the same main features (objects, type safety, exceptions, threads), all of which also came from the same tradition (and I believe they read our tech reports carefully ...). In addition, Java initially had innovations in bytecode verification and Web applets, and later had the full support of a large company, while we were only

supporting MODULA-3 from a research lab. I believe the module system in MODULA-3 is still vastly superior to programs such as JAVA, and that may explain continued interest.

Do you still use Modula-3 today? Is the language still being contributed to and updated?

While MODULA-3 was my all-time favorite language, I stopped using it after leaving DEC. I used JAVA for a short period, and today I occasionally use C# and F#.

How do you feel about statements such as this in Wikipedia: '*Modula-3 is now taught in universities only in comparative programming language courses, and its textbooks are out of print*' ?

It's probably accurate!

According to Wikipedia, the Modula-3 '*standard libraries [were] formally verified not to contain various types of bugs, including locking bugs.*' Why was this?

Type safety gets rid of a lot of silly bugs, but the main class of bugs it does not prevent are concurrency bugs. The expectation for MODULA-3 libraries was that they would have a complete description (in English), of exactly what each procedure did and what it required. There was social pressure at the time to make these descriptions very precise. Some were so precise that they were amenable to formal verification. This was considered important for some base libraries, particularly in terms of locking behavior, because locking bugs were not captured by the type system, and were the hardest to debug.

In your opinion, what lasting legacy has Modula-3 brought to computer development?

I think what's important is that MODULA-3 played a major role in popularizing the notion of type-safe programming. Cedar/Mesa was tremendously innovative, but was always kept secret at Xerox (I doubt that even now you can get its manual). And ML (the other root language of type safety) was always an academic non-object-oriented language. MODULA-3 was the stepping stone from Cedar/Mesa to JAVA; and today, type-safe programming is a given. I am personally very proud (as a former ML type-safe programmer) that I was able to hang-on to MODULA-3 until JAVA came out, therefore avoiding the C++ era altogether!

What are you proudest of in terms of the language's development and use?

The development of the type system, and the module system. In terms of use, we used it for over 10 years (including MODULA-2+) to write all our software, from OS's to GUI's, for several million lines of code. One of the most amazing features of MODULA-3 was the Network Objects, (but that was not my work), which was transferred directly to become JAVA RMI.

Where do you see computer programming languages heading in the future, particularly in the next 5 to 20 years?

Functional programming is coming back. Even an object-oriented language like C# now is a full functional language, in the sense that it supports first-class nameless lambda abstractions with proper scope capture and type inference, and developers love it. Other proper functional languages (which do not include object-oriented features) like F# and HASKELL are becoming more and more popular.

Do you have any advice for up-and-coming programmers?

Read other people's code!

Is there anything else of interest that you'd like to add?

Only that the most exciting MODULA-3 design meeting ever was abruptly interrupted by the San Francisco 7.1 earthquake.

Perl: Larry Wall

This time we chat with Larry Wall, creator of the Perl *programming language and regarded as the father of modern scripting languages*

What prompted the development of Perl?

I was scratching an itch, which is the usual story. I was trying to write reports based on text files and found the Unix tools were not quite up to it, so I decided I could do better. There was something missing in Unix culture – it was either C or a shell script, and people see them as opposites in one continuum. They were sort of orthogonal to each other and that is the niche Perl launched itself into – as a glue language. Unlike academic languages, which tend to be insular, I determined from the outset I was going to write Perl with interfaces.

Only later did it turn into a tool for something that was not anticipated. When the Web was invented they needed to generate text and use a glue language to talk to databases.

Was there a particular problem you were trying to solve?

You can tell the other problem by the reaction Perl got from the die hards in the Unix community. They said tools should do one thing and do them well. But they didn't understand Perl was not envisioned as a tool so much as a machine shop for writing tools.

How did the name Perl come about?

I came up with the name as I wanted something with positive connotations. The name originally had an 'a' in it. There was another lab stats language called Pearl, so I added another backronym. The second one is Pathologically Eclectic Rubbish Lister.

Do you ever find yourself using the 'backronym' Practical Extraction and Report Language at all?

It is meant to indicate that there is more than one way to do it, so we have multiple backronyms intentionally.

Were there any particularly hard/annoying problems you had to overcome in the development of the language?

The annoying thing when you're coming up with a new language is you can't really design it without taking into account the cultural context. A new language that violates everyone's cultural expectations has a hard time being accepted. Perl borrowed many aspects out of C, shell and AWK which were occasionally difficult to reconcile. For example, the use of $ in a regular expression might mean match a string or interpret a variable.

Would you have done anything differently in the development of Perl if you had the chance?

Either nothing or everything. See Perl 6.

What is the most interesting program that you've seen written with Perl?

I've seen an awful lot of interesting things written in Perl, maybe they are all weird. I know it's being used at the South Pole. The latest group to use it heavily are the biologists who do genetic analysis.

Have you ever seen the language used in a way that was not originally intended? If so, what was it? And did it work?

When Clearcase (revision control systems) wrote its device driver in Perl to access the file system underneath the kernel. The first surprising thing is that it worked. And the second surprising thing is that it was 10 times faster than their C code. Generally you would not want to write device drivers in Perl. Perl 6 maybe, but not Perl 5.

Has the evolution and popularity of the language surprised you in any way?

Yes and no. I definitely had experience prior to this with releasing open source software and finding that people liked it, so I already knew that if I wrote a language I liked other people

would probably like it too.

I didn't anticipate the scale of the acceptance over time. Perl 5 opened up to community development, and the best thing about Perl is CPAN.

In what way do you think employing natural language principles in Perl has contributed to it's success?

That's a subject of a PhD dissertation. We use natural language – most people think COBOL – and that's not how we think about it. Rather, the principles of natural language are that everything is context sensitive and there is more than one way to say it. You are free to learn it as you go.

We don't expect a five-year-old to speak with the same diction as a 50 year-old. The language is built to evolve over time by the participation of the whole community. Natural languages use inflection and pauses and tone to carry meanings. These carry over to punctuation in written language, so we're not afraid to use punctuation either.

What are you proudest of in terms of the language's initial development and continuing use?

It has to be the people involved. I think the Perl community paints a little picture in heaven. At the first Perl conference we met many in the Perl community for the first time and it was near my house so the family threw a party. The first thing we noticed is how pathologically helpful the people are and yet everyone was accepting of everyone's differences. The community counts diversity as a strength and that's what holds us together.

How did a picture of a camel come to end up on Programming Perl and consequently end up as a symbol for the language? Were you involved with this at all?

Yes. I picked the camel. When a writer writes a book for O'Reilly they ask them to suggest an animal. And then they say 'no, you are going to use a mongoose instead.'

If I had asked for a left-brain cover I would have asked for an oyster. But I shared the vision of the cover designer. The right-brain meaning of a camel is an animal self-sufficient in a dry place, and there are vague biblical connotations of a caravan. Since that was more or less the Perl bible for many years, it kind of naturally became the mascot.

Do you agree with statements stating that Perl is practical rather than beautiful? Was this your intention starting out?

Considering I wrote that into the first Perl manual page, yes. We are trying to make it more beautiful these days without loosing its usefulness. Maybe the next Perl book will have a camel with butterfly wings on it or something.

Many sources quote a main reference point of Perl being the C language. Was this deliberate?

Definitely. C has never exactly been a portable language but it is ubiquitous. By writing complex shell scripts and many macros you can create a portable C and then write a portable language on top. So that made Perl able to be ported everywhere and that was important to us.

How do you feel about the Comprehensive Perl Archive Network (CPAN) carrying over 13,500 modules by over 6,500 authors? Why do you think that the archive network has been such a success?

By it's very size it doesn't matter about Sturgeons Law – that 90 of everything is crud. 10 percent of a very large number is still a large number.

Do you agree with the following statement from Wikipedia: 'emThe design of Perl can be understood as a response to three broad trends in the computer industry: falling hardware costs, rising labour costs, and improvements in compiler technology. Many earlier computer languages, such as Fortran and C, were designed to make efficient use of expensive computer hardware. In contrast, Perl is designed to make efficient use of expensive computer programmers.'

That's accurate. In addition to C, I used YACC which was available. But I wrote my own lexer.

Do you agree that Perl is the 'duct tape of the Internet?'

It's one metaphor that we accept, but we like lots of metaphors.

Do you endorse the version of Perl written for Windows: win32.perl.org by Adam Kennedy?

Yes, I've used it and it is a good port.

You once listed the three virtues of a programmer as laziness, impatience and hubris. a) In what way do you think these virtues can be fostered by the way a language is designed, or are they merely a characteristic of a developer? b) How does the design of Perl encourage those virtues?

If you are lazy you look for shortcuts. If you are impatient you want your program to be done now. And as for the hubris, that makes the programs easier to distribute. That will help programs be used universally and that has some ego value.

My own take on that personally is it's been a privilege not to just write a programming language but invent a new medium of art that other people can work in.

Why has no specification or standard for the language been created?

There has for Perl 6. It's one of the things we decided to change. There will be multiple implementations of Perl 6, so it needs a standard and there will be a test suite.

We have a saying: all is fair if you pre-declare it. The idea with Perl 6 is you start with a standard language and you can mutate it. As long as you follow that refinement process there isn't the problem of ambiguity. There is the problem of multiple dialects, but that will always be a problem.

Have you ever played Perl Golf or written a Perl poem?

I wrote the first Perl poem and played Perl Golf. But I'm more well known for my obfuscation of C rather than Perl. It's been a long time.

What new elements does Perl 5.10.0 bring to the language? In what way is it preparing for Perl 6?

Perl 5.10.0 involves backporting some ideas from Perl 6, like switch statements and named pattern matches. One of the most popular things is the use of 'say' instead of 'print.'

This is an explicit programming design in Perl – easy things should be easy and hard things should be possible. It's optimised for the common case. Similar things should look similar but similar things should also look different, and how you trade those things off is an interesting design principle.

Huffman Coding is one of those principles that makes similar things look different.

And what about Perl 6? Do you have a release date for this yet? Are you able to talk about the most exciting/interesting new developments with this?

Sure, it's Christmas Day – we just don't say which one. We've been working on it 8 years now and we would like to think we are a lot closer to the end than the beginning. We're certainly well into the second 80 percent.

In your opinion, what lasting legacy has Perl brought to computer development?

An increased awareness of the interplay between technology and culture. Ruby has borrowed a few ideas from Perl and so has PHP. I don't think PHP understands the use of signals, but all languages borrow from other languages, otherwise they risk being single-purpose languages. Competition is good.

It's interesting to see PHP follow along with the same mistakes Perl made over time and recover from them. But Perl 6 also borrows back from other languages too, like Ruby. My ego may be big, but it's not that big.

Where do you envisage Perl's future lying?

My vision of Perl's future is that I hope I don't recognise it in 20 years.

Where do you see computer programming languages heading in the future, particularly in the next 5 to 20 years?

Don't design everything you will need in the next 100 years, but design the ability to create things we will need in 20 or 100 years. The heart of the Perl 6 effort is the extensibility we have built into the parser and introduced language changes as non-destructively as possible.

Do you have any advice for up-and-coming programmers?

We get Google Summer of Code people and various students are interested in the Perl 6 effort. We try to find a place for them to feel useful and we can always use people to write more tests.

In the future a lot of people will be getting into programming as a profession, but not calling it programming. They will call it writing spreadsheets or customising actions for their avatars. For many people it will be a means to an end, rather than an end in itself.

Python: Guido van Rossum

We chat with Guido van Rossum, Monty Python and Hitchhikers Guide to the Galaxy fan. Van Rossum is best known as the author of Python, and currently works for Google, CA where he gets to spend at least half his time developing the language

What was the motivation behind the development of such a productive programming language?

Long ago, around 1989, at CWI in Amsterdam, I was part of a group developing a novel operating system. We found that we needed to write a lot of applications to support users, and that writing these in C our productivity was atrocious. This made me want to use something like ABC, a language I had help implemented (also at CWI) earlier that decade.

ABC had much higher productivity than C, at the cost of a runtime penalty that was often acceptable for the kind of support applications we wanted to write: things that run only occasionally, for a short period of time, but possibly using complex logic. However, ABC had failed to gain popularity, for a variety of reasons, and was no longer being maintained (although you can still download it from `http://homepages.cwi.nl/~steven/abc`). It also wasn't directly usable for our purpose – ABC had been designed more as a teaching and data manipulation language, and its capabilities for interacting with the operating system (which we needed) were limited to non-existent by design.

Being youthful at the time I figured I could design and implement a language 'almost, but not quite, entirely unlike' ABC, improving upon ABC's deficiencies, and solve our support applications problem, so around Christmas 1989, I started hacking. For various reasons, soon after Python was complete enough to be used, that particular project was no longer as relevant, but Python proved useful to other projects at CWI, and in early 1991 (i. e. a little over a year after I started) we did the first open source release (well before the term open source had even been invented).

Was there a particular problem you were trying to solve?

Programmer productivity. My observation at the time was that computers were getting faster and cheaper at an incredible rate. Today this effect is of course known as Moore's law. At the same time, as long as the same programming languages were being used, the cost of the programmers to program them was not going down. So I set out to come up with a language that made programmers more productive, and if that meant that the programs would run a bit slower, well, that was an acceptable trade-off. Through my work on implementing ABC I had a lot of good ideas on how to do this.

Are you a Monty Python fan (given the name and other elements of the language derive from Monty Python's Flying Circus)?

Yes, this is where I took the language's name. The association with the snake of the same name was forced upon me by publishers who didn't want to license Monty-Python artwork for their book covers. I'm also into the Hitchhiker's Guide to the Galaxy, though I'm not into much other staples of geek culture (e. g. no sci-fi or fantasy, no role playing games, and definitely no computer gaming).

Given that the language was developed in the 1980s, what made you publish it in 1991?

I actually didn't start until the very end of 1989. It took just a bit over a year to reach a publishable stage.

Were there any particularly difficult or frustrating problems you had to overcome in the development of the language?

I can't remember anything particularly frustrating or difficult, certainly not during the first few years. Even management (usually the killer of all really interesting-but-out-of-field-left projects)

indulged my spending an inordinary amount of my time on what was considered mostly a hobby project at the time.

Would you do anything differently if you had the chance?

Perhaps I would pay more attention to quality of the standard library modules. Python has an amazingly rich and powerful standard library, containing modules or packages that handle such diverse tasks as downloading Web pages, using low-level Internet protocols, accessing databases, or writing graphical user interfaces. But there are also a lot of modules that aren't particularly well thought-out, or serve only a very small specialized audience, or don't work well with other modules.

We're cleaning up the worst excesses in Python 3.0, but for many reasons it's much harder to remove modules than to add new ones – there's always someone who will miss it. I probably should have set the bar for standard library modules higher than I did (especially in the early days, when I accepted pretty much anything anyone was willing to contribute).

A lot of current software is about writing for the Web, and there are many frameworks such as Django and Zope. What do you think about current Web frameworks based on Python?

For a few years there were definitely way too many Web frameworks. While new Web frameworks still occasionally crop up, the bar has been set much higher now, and many of the lesser-known frameworks are disappearing. There's also the merger between TurboGears and Pylons.

No matter what people say, Django is still my favorite – not only is it a pretty darn good Web framework that matches my style of developing, it is also an exemplary example of a good open source project, run by people who really understand community involvement.

What do you think about Ruby on Rails?

I've never used it. Obviously it's a very successful Web framework, but I believe (based on what users have told me) that Django is a close match.

We've all heard about how Python is heavily used by Google currently. How do you feel about this? Has this exceeded your expectations for the language?

I never had any specific expectations for Python, I've just always been happy to see the user community grow slowly but steadily. It's been a great ride.

Why has the language not been formally specified?

Very few open source languages have been formally specified. Formal language specifications seem to be particularly attractive when there is a company that wants to exercise control over a language (such as for Java and JavaScript), or when there are competing companies that worry about incompatible implementations (such as for C++ or SQL).

What's the most interesting program you've seen written with Python?

In terms of creative use of the language in a new environment, I think that would be MobilLenin, an art project for Nokia phones written by Jurgen Scheible.

Have you ever seen the language used in a way that wasn't originally intended?

Well, originally I had a pretty narrow view on Python's niche, and a lot of what people were doing with it was completely unexpected. I hadn't expected it to be used for writing expert systems, for example, and yet one of the early large examples was an expert system. I hadn't planned for it to be used to write high-volume network applications like Bittorrent either, and a few years back someone wrote a VOIP client in Python.

I also hadn't foreseen features like dynamic loading of extension modules, or using Python as an embedded programming language. And while ABC was designed in part as a teaching language, that was not a goal for Python's design, and I was initially surprised at Python's success in this area – though looking back I really should have expected that.

How do you feel about the title bestowed on you by the Python community: Benevolent Dictator for Life (BDFL)?

It totally matches the whimsical outlook I try to maintain on Python. By the way, the original

title (as I have recently rediscovered after digging through age-old email), invented in 1995, was First Interim Benevolent Dictator For Life. At a meeting of PYTHON developers and fans in Reston, Virginia, everyone present was bestowed with a jocular title, but mine was the only one that stuck.

Do you agree with the following statement taken from Wikipedia: '*Python can also be used as an extension language for existing modules and applications that need a programmable interface. This design, of a small core language with a large standard library and an easily-extensible interpreter, was intended by Van Rossum from the very start, due to his frustrations with ABC, which espoused the opposite mindset*'?

Yeah, that nails it. ABC was designed as a diamond – perfect from the start, but impossible to change. I realized that this had accidentally closed off many possible uses, such as interacting directly with the operating system: ABC's authors had a very low opinion of operating systems, and wanted to shield their users completely from all their many bizarre features (such as losing data when you removed a floppy disk at the wrong time). I didn't have this same fear: after all, PYTHON originated in an operating systems research group! So instead I built extensibility into the language from the get-go.

Do you believe that the large standard library is one of Python's greatest strengths?

Despite the misgivings about the quality of (parts of) the standard library that I expressed above, yes, very much so. It has often been a convincing argument for deciding to use PYTHON in a particular project when there were already standard library modules or packages to perform important tasks of the project at hand. Of course, the many third party extensions also add to this argument, but often it helps to notice that a single install (or, on modern Unix systems, no install at all, since PYTHON comes pre-installed) is all what's needed to get started.

Given that you launched the Computer Programming for Everybody (CP4E) initiative while working at the Corporation for National Research Initiatives (CNRI), and the clean syntax of Python, do you think that computer programming is an area that should be more accessible to the general public?

I certainly believe that educators would be wise to teach more about computer use than how to write PowerPoint presentations and HTML (useful though those are). Some people have been quite successful using PYTHON for all kinds of educational purposes, at many different levels. However education is an incredibly politicized subject and I've burned myself enough that I will refrain from further commentary on this subject.

How have Python Enhancement Proposals (PEPs) helped in the development of Python? Which is your favourite?

PEPs have made a huge difference. Before we started using PEPs, there was no specific process for getting something about the language or standard library changed: we had heated debates on the mailing list, but no clear way to make a decision, and no policy about what kinds of changes would need what kind of consensus. Sometimes people 'got lucky' by sending me a patch, and if I happened to like it, it went in – even though perhaps it wasn't always a well-thought-out design.

Other times good ideas got stuck in endless 'bikeshedding' (as it has now become known) about itty-bitty details. The PEP process helped frame these debates: the goal of a discussion was to arrive at a PEP, and a PEP needed to have a motivation, a specification, a discussion of alternatives considered and rejected, and so on. The PEP process (with slight variations) was also adopted by other open source projects. I'm sure this has helped generations of open source developers be more productive in their design discussions, and by having the whole PEP process written out (in PEP number 1) it also has served as education for new developers.

My favorite is PEP 666, which was written with the explicit objective to be rejected: it proposes a draconian attitude towards indentation, and its immediate rejection once and for all settled an argument that kept coming up (between tab-lovers and tab-haters). It is a great example of the rule that negative results are useful too.

Do you have any idea how many PEPs have been submitted over the language's history?

Yes, 239. While they aren't numbered consecutively, they are all kept under version control (at the same Subversion server we use for the PYTHON source tree) so they are easy to count. The highest-numbered one is 3141. This isn't counting a number of proposals that were nipped in the bud – occasionally someone drafts a PEP but before they can submit it for review it's already killed by the online discussion.

How has 3.01b been received since it's release in June this year? Does it vary greatly from the 3.01a release?

It's been received well – people are definitely downloading the successive betas (two so far with a third planned) and kicking the tyres. Perhaps the most visible difference from the last of the alphas is the standard library reorganization – that project hadn't really gotten started until the first beta. Other than that the differences are mostly minor improvements and bugfixes, nothing spectacular.

Do you currently use CPython?

It's the only PYTHON version I use regularly. It's also embedded in Google App Engine, and I use that a lot of course.

How do you feel about the 3.0 release series breaking backward compatibility?

It's the right think to do. There were a number of design problems in the language that just couldn't be fixed without breaking compatibility. But beyond those and a few cleanups we're actually trying not to break compatibility that much – many proposals to add new features that would introduce incompatibilities were rejected for that very reason, as long as an alternative was available that avoided the incompatibility.

Do you consider yourself a Pythonista?

It's not a term I would use myself, but if someone writes an email starting with 'Dear PYTHON' I certainly will assume I'm included in that audience. The Monty Python folks are sometimes referred to as Pythons; that's a term we never use. Similarly, Pythonesque tends to refer to 'in the style of Monty Python' while we use Pythonic meaning roughly 'compatible with PYTHON's philosophy.' Obviously that's a pretty vague term that is easily abused.

Where do you see Python going in the embedded space?

I'm assuming you're referring to platforms like cell phones and custom hardware and such. I think those platforms are ready for PYTHON, with enough memory and speed to comfortably run an interpreted language like PYTHON. (Hey, the PYTHON runtime is a lot smaller than the JVM!) Actual adoption differs – there's the Nokia S60 platform which has adopted PYTHON as its official scripting language, and embedded platforms running some form of Linux can in theory easily run PYTHON.

In your opinion, what lasting legacy has Python brought to computer development?

It has given dynamic languages a morale boost. It has shown that there are more readable alternatives to curly braces. And for many it has brought fun back to programming!

Where do you envisage Python's future lying?

Sorry, my crystal ball is in the shop, and I only use my time machine to go back in time to add features that are only now being requested. (That I have a time machine and use it for that purpose is a standing joke in the PYTHON community.)

Has the evolution and popularity of the language surprised you in anyway?

I certainly hadn't expected anything of the sort when I got started. I don't know what I expected though – I tend not to dwell too much on expectations and just like to fix today's problems. It also has come very gradually, so any particular milestone was never a big surprise. But after nearly 19 years I'm certainly very happy with how far we've come!

What are you proudest of in terms of the language's initial development and continuing use?

That PYTHON is and has always been the #1 scripting language at Google, without contest. Also, that the language has made it to the top 5 of dynamic languages on pretty much a zero PR budget. That's a tremendous achievement for a grassroots community.

Where do you see computer programming languages heading in the near future?

I hope that at some point computers will have sufficient power that we don't need separate functional, dynamic, and statically typed languages, and instead can use a single language that combines the benefits of all three paradigms.

Do you have any advice for up-and-coming programmers?

Learn more than one language. It's amazing how eye-opening it can be to compare and contrast two languages.

And finally, no interview on Python would be complete without the following questions: *a*. How do you feel about the indentation in Python now?

It's the right thing to do from a code readability point of view, and hence from a maintenance point of view. And maintainability of code is what counts most: no program is perfect from the start, and if it is successful, it will be extended. So maintenance is a fact of life, not a necessary evil.

***b*. Do you favour tabs or spaces?**

Definitely spaces. Four to be precise (even though the Google style guide uses two).

Is there anything else you'd like to add?

Hardly; you've been very thorough. I'd like to say hi to my many Aussie fans, and I promise that one of these years I'll be visiting your country to give some talks and do a bit of snorkeling. :-)

Scala: Martin Odersky

Scala is one of the newer languages that run on the Java Virtual Machine, which has become increasingly popular. Martin Odersky tells us about Scala's history, its future and what makes it so interesting

Why did you call the language Scala?

It means scalable language in the sense that you can start very small but take it a long way. For newcomers, it looks a bit like a scripting language. For the last two years we have actually been invited to compete in the JavaOne ScriptBowl, a Java scripting language competition. But Scala is not really a scripting language – that's not it's main characteristic. In fact, it can express everything that Java can and I believe there are a lot of things it can offer for large systems that go beyond the capabilities of Java. One of the design criteria was that we wanted to create a language that can be useful for everything from very small programs right up to huge systems and without the need to change structure along the way.

What led you to develop Scala?

In the 90s I became involved in the development of the Java language and its compiler. I got together with another researcher, Philip Wadler, and we developed *Pizza* that eventually led to *Generic Java (GJ)*, and then to Java version 5. Along the way I got to write the `javac` compiler. The compiler for GJ, which was our extension, was adopted as a standard long before Sun decided to adopt the GJ language constructs into Java – they took the compiler first.

When I moved to Switzerland 10 years ago I started to work on more fundamental topics. I did some research experiments to see if we could usefully combine functional and object-oriented programming. We had tried that already in 95/96 with *Pizza*, but that was only a half way success because there were a lot of rough edges, which all had to do with the fact that at the time we used Java as our base language. Java was not that malleable. So starting around 2000, I developed with my group at EPFL (Ecole Polytechnique Fédérale de Lausanne) new languages that would continue to inter-operate with Java but that would usefully combine object-oriented and functional programming techniques.

The first of these was called *Funnel* and the second was called Scala. The second experiment worked out pretty well, so we decided to wrap up the experimental phase and turn Scala into a real production language that people could rely on. We polished some edges, did some minor syntax changes, rewrote the Scala tools in Scala to make sure that the language and its tools could sustain heavy usage. Then we released Scala version 2 in 2006. It's been rapidly gaining popularity since then.

What are the main benefits of combining object-oriented and functional programming techniques?

They both bring a lot to the table. Functional programming lets you construct interesting things out of simple parts because it gives you powerful combinators – functions that take elements of your program and combine them with other elements in interesting ways. A related benefit of functional programming is that you can treat functions as data. A typical data type in almost all programming languages is 'int': you can declare an 'int' value anywhere, including inside a function, you can pass it to a function, return it from a function or store it in a field. In a functional language, you can do the same thing with functions: declare them inside other functions, pass them into and from functions, or store them in fields. These features give you a powerful way to build your own control structures, to define truly high-level libraries, or to define new domain specific languages.

Object-oriented programming, on the other hand, offers great ways to structure your system's components and to extend or adapt complicated systems. Inheritance and aggregation give you flexible ways to construct and organise your namespaces. There's good tool support like context help in IDEs (Integrated Development Environments) that will give you pop-up menus with all the methods that you can call at a given point.

The challenge was to combine the two so that it would not feel like two languages working side by side but would be combined into one single language.

I imagine the most significant part of that challenge was in deciding what to leave out?

Yes, if you took each language style in its entirety and combined them, you would end up with a lot of duplication and you would just have two sub-languages with little interaction between them. The challenge was to identify constructs from one side with constructs from the other. For instance, a function value in a functional programming language corresponds to an object in an object-oriented language. Basically you could say that it is an object with an 'apply' method. Consequently, we can model function values as objects.

Another example is in the algebraic data types of functional languages that can be modelled as class hierarchies on the object-oriented side. Also, the static fields and methods as found in JAVA. We eliminated these and modelled them with members of singleton objects instead. There are many other cases like these, where we tried to eliminate a language construct by matching and unifying it with something else.

What has been the overall greatest challenge you have faced in developing Scala?

Developing the compiler technology was definitely a challenge. Interestingly, the difficulties were more on the object-oriented side. It turned out that object-oriented languages with advanced static type systems were quite rare, and none of them were mainstream. SCALA has a much more expressive type system than JAVA or similar languages, so we had to break new ground by developing some novel type concepts and programming abstractions for component composition. That led to a quite a bit of hard work and also to some new research results.

The other hard part concerned interoperability. We wanted to be very interoperable so we had to map everything from JAVA to SCALA. There's always that tension between wanting to map faithfully the huge body of JAVA libraries while at the same time avoiding duplicating all constructs of the JAVA language. That was a persistent and challenging engineering problem. Overall I am quite happy with the result, but it has been a lot of work.

There are a lot of positive comments on forums about Scala's efficiency and scalability but another thing people often mention is that it is a very fun language to use. Was that also one of your aims in designing this language?

Absolutely. My co-workers and I spend a lot of time writing code so we wanted to have something that was a joy to program in. That was a very definite goal. We wanted to remove as many of the incantations of traditional high-protocol languages as possible and give SCALA great expressiveness so that developers can model things in the ways they want to. While writing `javac` I did a lot of JAVA programming and realised how much wasted work JAVA programmers have to do. In SCALA we typically see a two to three times reduction in the number of lines for equivalent programs. A lot of boilerplate is simply not needed. Plus it's a lot more fun to write.

This is a very powerful tool that we give to developers, but it has two sides. It gives them a lot of freedom but with that comes the responsibility to avoid misuse. Philosophically, I think that is the biggest difference between SCALA and JAVA. JAVA has a fairly restrictive set of concepts so that any JAVA program tends to look a bit like every other JAVA program and it is claimed that this makes it easy to swap programmers around. For SCALA, there's no such uniformity, as it is a very expressive programming language.

You can express SCALA programs in several ways. You can make them look very much like JAVA programs which is nice for programmers who start out coming from JAVA. This makes it very easy for programming groups to move across to SCALA, and it keeps project risks low. They can take a non-critical part first and then expand as fast as they think is right for them.

But you can also express SCALA programs in a purely functional way and those programs can end up looking quite different from typical JAVA programs. Often they are much more concise. The benefit that gives you is that you can develop your own idioms as high-level libraries or domain specific languages embedded into SCALA. Traditionally, you'd have to mix several different languages or configuration notations to achieve the same effect. So in the end, SCALA's single-language approach might well lead to simpler solutions.

The learning curve for a Java developer wanting to use Scala would be quite small but how easy would it be for programmers used to working with dynamic languages with dynamic disciplines such as PHP and Python and Ruby to use?

Clearly it is easiest for a Java or .NET developer to learn Scala. For other communities, the stumbling blocks don't have so much to do with the language itself as with the way we package it and the way the tools are set up, which is Java-specific. Once they learn how these things are set up, it should not be hard to learn the language itself.

What are your thoughts on Twitter using Scala? Is it good for the language's development that such a high profile site is using it?

That was great news. I am happy that they turned to Scala and that it has worked out well for them. Twitter has been able to sustain phenomenal growth, and it seems with more stability than what they had before the switch, so I think that's a good testament to Scala. When a high profile site like Twitter adopts a new language, it is really an acid test for that language. If there would be major problems with that language they'd probably be found rather quickly and highlighted prominently.

There are also a lot of other well-known companies adopting Scala. Sony Pictures Imageworks is using Scala to write its middle-tier software and Europe's largest energy company EDF is using Scala for contract modelling in its trading arm. SAP and Siemens are using Scala in their open source Enterprise Social Messaging Experiment (ESME) tool. That's just three examples of many.

One of Twitter's developers, Alex Payne, was saying that Scala could be the language of choice for the modern web start-up and could be chosen over other languages like Python and Ruby, which have been very popular but they are not as efficient as Scala. Do you agree and did you have Web 2.0 start-ups in mind while developing Scala?

I think Scala does make sense in that space. Twitter is not the only company who has realised this; LinkedIn also uses Scala.

I think what Scala offers there is the ability to build on a solid high performance platform – the Java Virtual Machine (JVM) while still using an agile language. There are some other options that fall into that category such as Jython, JRuby, Groovy, or Clojure, but these are all dynamically typed languages on the JVM.

In the end the question comes down to whether you are more comfortable in a statically typed setting, be it because that will catch many errors early, because it gives you a safety net for refactorings, or because it helps with performance. Or you may think you need a fully dynamic language because you want to do fancy stuff with metaprogramming. In the end it comes down to that choice. If you prefer a statically typed language, I think Scala is definitely the best option today.

What is your favourite feature of the language, if you had to pick?

I don't think I can name a single favourite feature. I'd rather pick the way Scala's features play together. For instance, how higher-order functions blend with objects and abstract types, or how actors were made possible because functions in Scala can be subclassed. The most interesting design patterns in Scala come precisely from the interaction between object-oriented and functional programming ideas.

Where do you see Scala headed in the future?

In the near term, we are currently working hard on the next release, Scala 2.8, where we are focusing on things like high performance array operations and re-defined collection libraries with fast persistent data structures, among others. That should be out by autumn this year.

Then in the long term we see interesting opportunities around concurrency and parallelism, so we are looking at new ways to program multicore processors and other parallel systems. We already have a head start here because Scala has a popular actor system which gives you a high-level way to express concurrency. This is used in Twitter's message queues, for instance. The interesting thing is that actors in Scala are not a language feature, they have been done

purely as a SCALA library. So they are a good witness to SCALA's flexibility: you can program things that will look like language features to application programmers by shipping the right kind of primitives and abstractions in a library.

We are hoping that what works for actors will also work for other concurrent abstractions such as data parallelism and stream programming. I think that in the future we will probably need several concurrency abstractions to really make use of multicore because different profiles of parallelism and concurrency will require different tools. I think that SCALA's library based approach is relevant here, because it lets us mix and match concepts implemented as SCALA classes and objects, thus moving forward quickly rather than having to put all of this into a language and a compiler. I think this work will keep us busy for the next four or five years.

Sh: Steve Bourne

In the early 1970s Bourne was at the Computer Laboratory in Cambridge, England working on a compiler for Algol 68 as part of his PhD work in dynamical astronomy. This work paved the way for him to travel to IBM's T. J. Watson Research Center in New York in 1973, in part to undertake research into compilers. Through this work, and a series of connections and circumstance, Bourne got to know people at Bell Labs who then offered him a job in the Unix group in 1975. It was during this time Bourne developed sh

What prompted the creation of the Bourne shell?

The original shell wasn't really a language; it was a recording – a way of executing a linear sequence of commands from a file, the only control flow primitive being goto a label. These limitations to the original shell that Ken Thompson wrote were significant. You couldn't, for example, easily use a command script as a filter because the command file itself was the standard input. And in a filter the standard input is what you inherit from your parent process, not the command file.

The original shell was simple but as people started to use Unix for application development and scripting, it was too limited. It didn't have variables, it didn't have control flow, and it had very inadequate quoting capabilities.

My own interest, before I went to Bell Labs, was in programming language design and compilers. At Cambridge I had worked on the language ALGOL 68 with Mike Guy. A small group of us wrote a compiler for ALGOL 68 that we called Algol68C. We also made some additions to the language to make it more usable. As an aside we bootstrapped the compiler so that it was also written in Algol68C.

When I arrived at Bell Labs a number of people were looking at ways to add programming capabilities such as variables and control flow primitives to the original shell. One day [mid 1975?] Dennis [Ritchie] and I came out of a meeting where somebody was proposing yet another variation by patching over some of the existing design decisions that were made in the original shell that Ken wrote. And so I looked at Dennis and he looked at me and I said 'you know we have to re-do this and re-think some of the original design decisions that were made because you can't go from here to there without changing some fundamental things.' So that is how I got started on the new shell.

Was there a particular problem that the language aimed to solve?

The primary problem was to design the shell to be a fully programmable scripting language that could also serve as the interface to users typing commands interactively at a terminal.

First of all, it needed to be compatible with the existing usage that people were familiar with. There were two usage modes. One was scripting and even though it was very limited there were already many scripts people had written. Also, the shell or command interpreter reads and executes the commands you type at the terminal. And so it is constrained to be both a command line interpreter and a scripting language. As the Unix command line interpreter, for example, you wouldn't want to be typing commands and have all the strings quoted like you would in C, because most things you type are simply uninterpreted strings. You don't want to type *`ls directory`* and have the directory name in string quotes because that would be such a royal pain. Also, spaces are used to separate arguments to commands. The basic design is driven from there and that determines how you represent strings in the language, which is as un-interpreted text. Everything that isn't a string has to have something in front of it so you know it is not a string. For example, there is `$` sign in front of variables. This is in contrast to a typical programming language, where variables are names and strings are in some kind of quote marks. There are also reserved words for built-in commands like for loops but this is common with many programming languages.

So that is one way of saying what the problem was that the Bourne Shell was designed to solve. I would also say that the shell is the interface to the Unix system environment and so that's its

primary function: to provide a fully functional interface to the Unix system environment so that you could do anything that the Unix command set and the Unix system call set will provide you. This is the primary purpose of the shell.

One of the other things we did, in talking about the problems we were trying to solve, was to add environment variables to Unix system. When you execute a command script you want to have a context for that script to operate in. So in the old days, positional parameters for commands were the primary way of passing information into a command. If you wanted context that was not explicit then the command could resort to reading a file. This is very cumbersome and in practice was only rarely used. We added environment variables to Unix. These were named variables that you didn't have to explicitly pass down from the parent to the child process. They were inherited by the child process. As an example you could have a search path set up that specifies the list of directories to use when executing commands. This search path would then be available to all processes spawned by the parent where the search path was set. It made a big difference to the way that shell programming was done because you could now see and use information that is in the environment and the guy in the middle didn't have to pass it to you. That was one of the major additions we made to the operating system to support scripting.

How did it improve on the Thompson shell?

I did change the shell so that command scripts could be used as filters. In the original shell this was not really feasible because the standard input for the executing script was the script itself. This change caused quite a disruption to the way people were used to working. I added variables, control flow and command substitution. The case statement allowed strings to be easily matched so that commands could decode their arguments and make decisions based on that. The for loop allowed iteration over a set of strings that were either explicit or by default the arguments that the command was given.

I also added an additional quoting mechanism so that you could do variable substitutions within quotes. It was a significant redesign with some of the original flavour of the Thompson shell still there. Also I eliminated `goto` in favour of flow control primitives like `if` and `for`. This was also considered rather radical departure from the existing practice.

Command substitution was something else I added because that gives you a very general mechanism to do string processing; it allows you to get strings back from commands and use them as the text of the script as if you had typed it directly. I think this was a new idea that I, at least, had not seen in scripting languages, except perhaps Lisp.

How long did this process take?

It didn't take very long; it's surprising. The direct answer to the question is about maybe 3-6 months at the most to make the basic design choices and to get it working. After that I iterated the design and fixed bugs based on user feedback and requests.

I honestly don't remember exactly but there were a number of design things I added at the time. One thing that I thought was important was to have no limits imposed by the shell on the sizes of strings or the sizes of anything else for that matter. So the memory allocation in the implementation that I wrote was quite sophisticated. It allowed you to have strings that were any length while also maintaining a very efficient string processing capability because in those days you couldn't use up lots of instructions copying strings around. It was the implementation of the memory management that took the most time. Bugs in that part of any program are usually the hardest to find. This part of the code was worked on after I got the initial design up and running.

The memory management is an interesting part of the story. To avoid having to check at run time for running out of memory for string construction I used a less well known property of the `sbrk` system call. If you get a memory fault you can, in Unix, allocate more memory and then resume the program from where it left off. This was an infrequent event but made a significant difference to the performance of the shell. I was assured at the time by Dennis that this was part of the `sbrk` interface definition. However, everyone who ported Unix to another computer found this out when trying to port the shell itself. Also at that time at Bell Labs,

there were other scripting languages that had come into existence in different parts of the lab. These were efforts to solve the same set of problems I already described. The most widely used 'new' shell was in the programmer's workbench – John Mashey wrote that. And so there was quite an investment in these shell scripts in other parts of the lab that would require significant cost to convert to the new shell.

The hard part was convincing people who had these scripts to convert them. While the shell I wrote had significant features that made scripting easier, the way I convinced the other groups was with a performance bake off. I spent time improving the performance, so that probably took another, I dont know, 6 months or a year to convince other groups at the lab to adopt it. Also, some changes were made to the language to make the conversion of these scripts less painful.

How come it fell on you to do this?

The way it worked in the Unix group [at Bell Labs] was that if you were interested in something and nobody else owned the code then you could work on it. At the time Ken Thompson owned the original shell but he was visiting Berkeley for the year and he wasn't considering working on a new shell so I took it on. As I said I was interested in language design and had some ideas about making a programmable command language.

Have you faced any hard decisions in maintaining the language?

The simple answer to that is I stopped adding things to the language in 1983. The last thing I added to the language was functions. And I don't know why I didn't put functions in the first place. At an abstract level, a command script is a function but it also happens to be a file that needs to be kept track of. But the problem with command files is one of performance; otherwise, there's not a lot of semantic difference between functions and command scripts. The performance issue arises because executing a command script requires a new process to be created via the Unix `fork` and `exec` system calls; and that's expensive in the Unix environment. And so most of the performance issues with scripting come from this cost. Functions also provide abstraction without having a `fork` and `exec` required to do the implementation. So that was the last thing I added to the language.

Any one language cannot solve all the problems in the programming world and so it gets to the point where you either keep it simple and reasonably elegant, or you keep adding stuff. If you look at some of the modern desktop applications, they have feature creep. They include every bell, knob and whistle you can imagine and finding your way around is impossible. So I decided that the shell had reached its limits within the design constraints that it originally had. I said 'you know there's not a whole lot more I can do and still maintain some consistency and simplicity.' The things that people did to it after that were make it POSIX compliant and no doubt there were other things that have been added over time. But as a scripting language I thought it had reached the limit.

Looking back, is there anything you would change in the language's development?

In the language design I would certainly have added functions earlier. I am rather surprised that I didn't do that as part of the original design. And the other thing I would like to have done is written a compiler for it. I got halfway through writing a shell script compiler but shelved it because nobody was complaining about performance at the time.

I can't think of things that we would have done particularly differently looking back on it. As one of the first programmable scripting languages it was making a significant impact on productivity.

If the language was written with the intention of being a scripting language, how did it become more popular as an interactive command interpreter?

It was designed to do both from the start. The design space was you are sitting at the terminal, or these days at the screen, and you're typing commands to get things done. And it was always intended that that be one of the primary functions of the shell. This is the same set of commands that you're accessing when you're in a shell script because you're (still) accessing the Unix environment but just from a script. It's different from a programming language in that

you are accessing essentially the Unix commands and those capabilities either from the terminal or from the script itself. So it was originally intended to do both. I have no idea which is more popular at this point; I think there are a lot of shell scripts around.

Many other shells have been written including the Bourne Again shell (Bash), Korn Shell (ksh), the C Shell (csh), and variations such as tcsh. What is your opinion on them?

I believe that BASH is an open source clone of the Bourne shell. And it may have some additional things in it, I am not sure. It was driven (I'm sure everybody knows this) from the open source side of the world because the Unix licence tied up the Unix intellectual property (source code) so you had to get the licence in order to use it.

The C shell was done a little after I did the Bourne shell – I talked to Bill Joy about it at the time. He may have been thinking about it at the same time as I was writing SH but anyway it was done in a similar time frame. Bill was interested in some other things that at the time I had less interest in. For example, he wanted to put in the history feature and job control so he went ahead and wrote the C shell. Maybe in retrospect I should have included some things like history and job control in the Unix shell. But at the time I thought they didn't really belong in there ... when you have a window system you end up having some of those functions anyway.

I don't recall exactly when the Korn shell was written. The early 80s I suspect. At the time I had stopped adding 'features' to SH and people wanted to continue to add things like better string processing. Also POSIX was being defined and a number of changes were being considered in the standard to the way SH was being used. I think KSH also has some CSH facilities such as job control and so on. My own view, as I have said, was that the shell had reached the limits of features that could be included without making it rather baroque and certainly more complex to understand.

Why hasn't the C shell (and its spawn) dropped off the edge of the planet? Is that actually happening?

I don't know, is it? There are a lot of scripts that people would write in the C shell. It has a more C-like syntax also. So once people have a collection of scripts then it's hard to get rid of it. Apart from history and job control I don't think the language features are that different although they are expressed differently. For example, both languages have loops, conditionals, variables and so on. I imagine some people prefer the C-style syntax, as opposed to the ALGOL 68-like syntax of the shell.

There was a reason that I put the ALGOL-like syntax in there. I always found, and this is a language design issue, that I would read a C program and get to a closing brace and I would wonder where the matching opening brace for that closing brace was. I would go scratching around looking for the beginning of the construct but you had limited visual clues as to what to look for. In the C language, for example, a closing brace could be the end of an `if` or `switch` or a number of other things. And in those days we didn't have good tools that would allow you to point at the closing brace and say 'where's the matching opening brace?'. You could always adopt an indenting convention but if you indented incorrectly you could get bugs in programs quite easily because you would have mismatching or misplaced brace. So that was one reason why I put in the matching opening and closing tokens like an `if` and a `fi` – so all of the compound statements were closed and had unique closing tokens.

And it was important for another reason: I wanted the language to have the property that anywhere where there was a command you could replace it with any closed form command like an `if-fi` or a `while-do-done` and you could make that transformation without having to go re-write the syntax of the thing that you were substituting. They have an easily identifiable start and end, like matching parentheses.

Compare current Unix shells (programs that manipulate text) and new MS Windows Power Shell (classes that manipulate objects). Would Unix benefit from a Power Shell approach?

The Unix environment itself doesn't really have objects if you look at what the shell is interfacing to, which is Unix. If objects are visible to the people writing at the shell level then it would need

to support them. But I don't know where that would be the case in Unix; I have not seen them. I imagine in the Microsoft example objects are a first class citizen that are visible to the user so you want to have them supported in the scripting language that interfaces to Windows. But that is a rather generic answer to your question; I am not specifically familiar with the power shell.

Is Bash a worthy successor to Bourne shell? Should some things in Bash have been done differently?

I believe you can write shell scripts that will run either in the Bourne shell or BASH. It may have some additional features that aren't in the Bourne shell. I believe BASH was intended as a strictly compatible open source version of the Bourne shell. Honestly I haven't looked at it in any detail so I could be wrong. I have used BASH myself because I run a GNU/Linux system at home and it appears to do what I would expect.

Unix specialist Steve Parker has posted *Steve's Bourne / Bash scripting tutorial* in which he writes: '*Shell script programming has a bit of a bad press amongst some Unix systems administrators. This is normally because of one of two things: a) The speed at which an interpreted program will run as compared to a C program, or even an interpreted Perl program; b) Since it is easy to write a simple batch-job type shell script, there are a lot of poor quality shell scripts around.*' Do you agree?

It would be hard to disagree because he probably knows more about it than I do. The truth of the matter is you can write bad code in any language, or most languages anyway, and so the shell is no exception to that. Just as you can write obfuscated C you can write obfuscated shell. It may be that it is easier to write obfuscated shell than it is to write obfuscated C. I don't know. But that's the first point.

The second point is that the shell is a string processing language and the string processing is fairly simple. So there is no fundamental reason why it shouldn't run fairly efficiently for those tasks. I am not familiar with the performance of BASH and how that is implemented. Perhaps some of the people that he is talking about are running BASH versus the shell but again I don't have any performance comparisons for them. But that is where I would go and look. I know when I wrote the original implementation of the shell I spent a lot of time making sure that it was efficient. And in particular with respect to the string processing but also just the reading of the command file. In the original implementation that I wrote, the command file was pre-loaded and pre-digested so when you executed it you didn't have to do any processing except the string substitutions and any of the other semantics that would change values. So that was about as efficient as you could get in an interpretive language without generating code.

I will say, and it is funny because Maurice Wilkes asked me this question when I told him what I was doing, and he said 'how can you afford to do that?' Meaning, how can you afford to write programs when the primitives are commands that you are executing and the costs of executing commands is so high relative to executing a function in a C program, for example. As I have said earlier, the primary performance limitation is that you have to do a Unix `fork` and `exec` whenever you execute a command. These are much more expensive than a C function call. And because commands are the abstraction mechanism, that made it inefficient if you are executing many commands that don't do much.

Where do you envisage the Bourne shell's future lying?

I don't know; it's a hard question. I imagine it will be around as long as Unix is around. It appears to be the most ubiquitous of the Unix shells. What people tell me is if they want one that is going to work on all the Unix systems out there in the world, they write it in the Bourne shell (or BASH). So, that's one reason. I don't know if it is true but that is what they tell me. And I don't see Unix going away any time soon. It seems to have had a revival with the open source movement, in particular the GNU Project and the Linux kernel.

Where do you see shells going in general?

As I have said the shell is an interface to the Unix environment. It provides you with a way of invoking the Unix commands and managing this environment interactively or via scripts. And

that is important because if you look at other shells, or more generally scripting languages, they typically provide access to, or control and manipulate, some environment. And they reflect, in the features that are available to the programmer, the characteristics of the environment they interface to. It's certainly true the Unix shells are like that. They may have some different language choices and some different trade offs but they all provide access to the Unix environment.

So you are going to see languages popping up and shells popping up. Look at some of the developments that are going on with the Web – a number of languages have been developed that allow you to program HTML and Web pages, such as PHP. And these are specific to that environment. I think you are going to see, as new environments are developed with new capabilities, scripting capabilities developed around them to make it easy to make them work.

How does it feel to have a programming language named after you?

People sometimes will say to me 'oh, you're Steve Bourne' because they are familiar with the shell. It was used by a lot of people. But you do a lot of things in your life and sometimes you get lucky to have something named after you. I don't know who first called it the Bourne shell.

I thought it was you that named it Bourne?

No. We just called it 'the shell' or 'sh.' In the Unix group back in the labs I wrote a couple of other programs as well, like the debugger `adb`, but we didn't call that 'the Bourne adb.' And certainly we didn't call it 'the Aho awk.' And we didn't call it 'Feldman make.' So I didn't call it the Bourne shell, someone else did. Perhaps it was to distinguish it from the other shells around at the time.

Where do you see computer programming languages heading in the future, particularly in the next 5 to 20 years?

You know I have tried to predict some of these things and I have not done very well at it. And in this business 20 years is an eternity. I am surprised at the number of new entrants to the field. I thought that we were done with programming language designs back in the late 70s and early 80s. And maybe we were for a while. We had C, C++ and then along comes Java and Python and so on. It seems that the languages that are the most popular have a good set of libraries or methods available for interfacing to different parts of the system. It is also true that these modern languages have learned from earlier languages and are generally better designed as a result.

Since I was wrong in 1980 when we thought 'well we are done with languages, let's move on to operating systems, object-oriented programming, and then networking' and whatever else were the other big problems at the time. And then suddenly we get into the Internet Web environment and all these things appear which are different and improved and more capable and so on. So it is fun to be in a field that continues to evolve at such a rapid pace.

You can go on the Internet now and if you want to write, for example, a program to sort your mail files, there is a Python or Perl library you will find that will decode all the different kinds of mail formats there are on the planet. You can take that set of methods or library of functions and use it without having to write all the basic decoding yourself. So the available software out there is much more capable and extensive these days.

I think we will continue to see specialised languages; such as PHP which works well with Web pages and HTML. And then look at Ruby on Rails. Who would have thought Lisp would come back to life. It is fun to be an observer and learn these new things.

Do you think there are too many programming languages?

Maybe. But the ones that are good will survive and the ones that aren't will be seen as fads and go away. And who knows at the time which ones are which. They are like tools in a way; they are applicable in different ways. Look at any engineering field and how many tools there are. Some for very specific purposes and some quite general.

The issue is 'What set of libraries and methods are available to do all the things you want to do?'. Like the example I gave about mail files. There are dozens of things like that where you want to be able to process certain kinds of data. And so you want libraries to do things. For

example, suppose you want a drawing package. And the question is: what do you want to use the drawing package for? If you are going to write programs to do that do you write them in PERL or PYTHON or what? So it is going to be driven as much by the support these languages have in terms of libraries and sets of methods they have as by the language itself.

If you were teaching up-and-coming programmers, what would you say?

First, I would be somewhat intimidated because they all know more than I do these days! And the environments today are so much more complicated than when I wrote code. Having said that software engineering hasn't changed much over the years. The thing we practised in the Unix group was if you wrote some code then you were personally accountable for that code working and if you put that code into public use and it didn't work then it was your reputation that was at stake. In the Unix lab there were about 20 people who used the system every day and we installed our software on the PDP 11 that everyone else was using. And if it didn't work you got yelled at rather quickly. So we all tested our programs as much as we could before releasing them to the group. I think that this is important these days – it's so easy in these large software projects to write code and not understand the environment you will be operating in very well, so it doesn't work when you release the code in the real world. That is, one piece of advice I'd give is to make sure you understand who is using your code and what they will use it for. If you can, go and visit your customers and find out what they are doing with your code. Also be sure to understand the environment that your program will be deployed into. Lastly, take pride in your code so that your peers and customers alike will appreciate your skill.

Tcl: John Ousterhout

Tcl creator John Ousterhout took some time to tell Computerworld about the extensibility of Tcl, its diverse eco-system and use in NASA's Mars Lander project

What prompted the creation of Tcl?

In the early and mid-1980's my students and I created several interactive applications, such as editors and analysis tools for integrated circuits. In those days all interactive applications needed command-line interfaces, so we built a simple command language for each application. Each application had a different command language, and they were all pretty weak (for example, they didn't support variables, looping, or macros). The idea for Tcl came to me as a solution to this problem: create a powerful command language, and implement it as a library package that can be incorporated into a variety of different applications to form the core of the applications' command languages.

Was there a particular problem the language aimed to solve?

The original goal for Tcl was to make it easy to build applications with powerful command languages. At the time I didn't envision Tcl being used as a stand-alone programming language, though that is probably the way that most people have used it.

How does Tk fit into the picture?

One of the key features of Tcl is extensibility: it is easy to create new features that appear as part of the language (this is the way that applications using Tcl can make their own functionality visible to users). At the same time that I was developing Tcl, graphical user interfaces were becoming popular, but the tools for creating GUI applications (such as the Motif toolkit for the X Window System) were complex, hard to use, and not very powerful. I had been thinking about graphical toolkits for several years, and it occurred to me that I could build a toolkit as an extension to Tcl. This became Tk. The flexible, string-oriented nature of Tcl made it possible to build a toolkit that was simple to use yet very powerful.

What influence, if any, did Tcl have in the development of Java?

As far as I know the Java language developed independently of Tcl. However, the AWT GUI toolkit for Java reflects a few features that appeared first in Tk, such as a grid-based geometry manager.

What's the Tcl eco-system like?

The Tcl ecosystem is so diverse that it's hard to characterize it, but it divides roughly into two camps. On the one hand are the Tk enthusiasts who believe that the Tcl/Tk's main contribution is its powerful cross-platform GUI tools; they think of Tcl/Tk as a stand-alone programming platform, and are constantly pushing for more Tk features. On the other hand are the Tcl purists who believe the most unique thing about Tcl is that it can be embedded into applications. This group is most interested in the simplicity and power of the APIs for embedding. The Tcl purists worry that the Tk enthusiasts will bloat the system to the point where it will no longer be embeddable.

What is Tcl's relevance in the Web application world?

One of my few disappointments in the development of Tcl is that it never became a major factor in Web application development. Other scripting languages, such as JavaScript and Python, have played a much larger role than Tcl.

What was the flagship application made with Tcl?

Tcl's strength has been the breadth of activities that it covers, rather than a single flagship application. Most Tcl applications are probably small ones used by a single person or group. At the same time, there are many large applications that have been built with Tcl, including the NBC broadcast control system, numerous applications in the Electronic Design Automation space, test harnesses for network routers and switches, Mars lander software, and the control

system for oil platforms in the Gulf of Mexico.

Unfortunately I don't know very much about those projects, and the information I have is pretty old (I heard about both of those projects in the late 1990s). For the oil platform project, I believe that Tcl/Tk provided the central management system for observing the overall operation of the platform and controlling its functions. In the case of the Mars lander, I believe Tcl was used for pre-launch testing of the system hardware and software.

Have you ever seen the language used in a way that wasn't originally intended?

The most surprising thing to me was that people built large programs with Tcl. I designed the language as a command-line tool and expected that it would be used only for very short programs: perhaps a few dozen lines at most. When I went to the first Tcl workshop and heard that a multi-billion-dollar oil platform was being controlled by a half million lines of Tcl code I almost fell over.

Were there any particularly difficult or frustrating problems you had to overcome in the development of the language?

One problem we worked on for many years was making Tcl and Tk run on platforms other than Unix. This was eventually successful, but the differences between Unix, Windows, and the Macintosh were large enough that it took a long time to get it all right. A second problem was language speed. Originally Tcl was completely interpreted: every command was reparsed from a string every time it was executed. Of course, this was fairly inefficient. Eventually, Brian Lewis created a bytecode compiler for Tcl that provided 5-10× speedups.

Can you attribute any of Tcl's popularity to the Tk framework?

Absolutely. As I mentioned earlier, there are many people who use Tcl exclusively for Tk.

Generally, more and more and more coding is moving to scripting languages. What do you think about this trend given Tcl's long scripting language heritage? Has Tcl gained from this trend?

I think this trend makes perfect sense. Scripting languages make it substantially easier to build and maintain certain classes of applications, such as those that do a lot of string processing and those that must integrate a variety of different components and services. For example, most Web applications are built with scripting languages these days.

Looking back, is there anything you would change in the language's development?

Yes, two things. First, I wish I had known that people would write large programs in Tcl; if I had, I'm sure I would have done some things differently in the design of the language. Second, I wish I had included object-oriented programming facilities in the language. I resisted this for a long time, and in retrospect I was wrong. It would have been easy to incorporate nice object-oriented facilities in Tcl if I had done it early on. Right now there are several Tcl extensions that provide OO facilities but there is not one 'standard' that is part of Tcl; this is a weakness relative to other scripting languages.

Where do you envisage Tcl's future lying?

Tcl is more than 20 years old now (hard to believe!) so it is pretty mature; I don't expect to see any shocking new developments around Tcl or Tk. I'm sure that Tcl and Tk will continue to be used for a variety of applications.

YACC: Stephen Johnson

Stephen C. Johnson, the inventor of of YACC, is an AT&T alumni and is currently employed at The MathWorks, where he works daily with MATLAB. Computerworld snatched the opportunity recently to get his thoughts on working with Al Aho and Dennis Ritchie, as well as the development of Bison

What made you name your parser generator in the form of an acronym: Yet Another Compiler-Compiler?

There were other compiler-compilers in use at Bell Labs, especially as part of the Multics project. I was familiar with a version of McClure's TMG. When Jeff Ullman heard about my program, he said in astonishment 'Another compiler-compiler?'. Thus the name.

What prompted the development of YACC? Was it part of a specific project at AT&T Labs?

'Project' sounds very formal, and that wasn't the Bell Labs way. The Computer Science Research group had recently induced AT&T to spend many million dollars on Multics, with nothing to say for it. Some of my co-workers felt that the group might be disbanded. But in general, Bell Labs hired smart people and left a lot of interesting problems around. And gave people years to do things that were useful. It's an environment that is almost unknown now.

YACC began for me as an attempt to solve a very simple, specific problem.

What problem were you trying to solve?

Dennis Ritchie had written a simple language, B, which ran on our GE (later Honeywell) system, and I started to use it to write some systems programs. When Dennis started to work on Unix, the compiler became an orphan, and I adopted it. I needed access to the exclusive-or operation on the computer, and B did not have any way to say that. So, talking to Dennis, we agreed what would be a good name for the operator, and I set out to put it into the compiler. I did it, but it was no fun.

One day at lunch I was griping about this, and Al Aho said 'There's a paper by Knuth – I think he has a better way.' So Al agreed to build the tables for the B expression grammar. I remember giving him about 30 grammar rules, and he went up to the stockroom and got a big piece of paper, about 2 by 3 feet, ruled it into squares, and started making entries in it. After an hour of watching him, he said 'this will take a while.' In fact, it took about 2 days!

Finally, Al handed me the paper in triumph, and I said 'what do I do with this?' He taught me how to interpret the table to guide the parser, but when I typed the table in and tried to parse, there were errors. Each error we found involved another hour of Al's time and some more rows in the table. Finally, after the third time I asked him 'what are you doing when you make the table?' He told me, and I said 'I could write a program to do that!' And I did.

Did you experience any particular problems in the development of YACC?

Especially after I moved to Unix, memory size and performance became an obsession. We had at most 64K bytes to hold the program and data, and we wanted to do FORTRAN.

When YACC first ran, it was very slow – it implemented Knuth's ideas very literally. A grammar with 50 rules took about 20 minutes to run, which made me very unpopular with my co-workers ('Damn, Johnson's running YACC again!'). I set out to improve the size and space characteristics. Over the next several years, I rewrote the program over a dozen times, speeding it up by a factor of 10,000 or so. Many of my speedups involved proving theorems that we could cut this or that corner and still have a valid parser. The introduction of precedence was one example of this.

Dennis was actively working on B while I was writing YACC. One day, I came in and YACC would not compile – it was out of space. It turns out that I had been using every single slot in the symbol table. The night before, Dennis had added the `for` statement to B, and the word 'for' took a slot, so YACC no longer fit!

While small memory was a major pain, it also imposed a discipline on us that removed mental

clutter from our programs, and that was a very good thing.

Would you do anything differently if you got the chance to develop YACC all over again?

I'd try harder to find a notation other than `$1`, `$2`, `$$`, etc. While simple and intuitive, the notation is a source of errors as grammars evolve.

What's the most interesting program you've seen that uses YACC?

Some of the most interesting uses I've seen came very early. Brian Kernighan was an early user when he wrote the EQN utility that typeset mathematical equations. And Mike Lesk wrote a grammar to try to parse English. Both grammars were highly ambiguous, with hundreds of conflicts. Al Aho used to break out in a rash when he contemplated them, but they worked fine in practice and gave me some very challenging practical applications of YACC.

Have you ever seen YACC used in a way that you didn't originally intend? If so, what was it? And did it or didn't it work?

Mike's use of YACC to parse English was one. He used the YACC tables, but wrote a parser that would keep multiple parses around simultaneously. It wasn't really that successful, because even rather simple sentences had dozens of legal parses. With 64K of memory to play with, there wasn't much he could do to resolve them.

How do you feel now that other programs such as Abraxas pcYACC and Berkeley YACC have taken over as default parser generators on Unix systems?

Actually, I'm amazed that YACC is still around at all after 35 years. It's a tribute to Knuth's insights. And I also have to say that the work I put into making YACC very fast and powerful kept it viable much longer that it otherwise would have been.

Did you foresee the development of Bison?

Given GNU's desire to replicate Unix, I think Bison was inevitable. I am bemused that some GNU people are so irritated that GNU's contribution to Linux is not recognized, but yet they have failed to recognize their debt to those of us who worked on Unix.

In your opinion, what lasting legacy has YACC brought to language development?

YACC made it possible for many people who were not language experts to make little languages (also called domain-specific languages) to improve their productivity. Also, the design style of YACC – base the program on solid theory, implement the theory well, and leave lots of escape hatches for the things you want to do that don't fit the theory – was something many Unix utilities embodied. It was part of the atmosphere in those days, and this design style has persisted in most of my work since then.

Where do you envisage the future of parser generators lying?

The ideas and techniques underlying YACC are fundamental and have application in many areas of computer science and engineering. One application I think is promising is using compiler-design techniques to design GUIs – I think GUI designers are still writing GUIs in the equivalent of assembly language, and interfaces have become too complicated for that to work any more.

What are you proudest of in terms of YACC's development and use?

I think computing is a service profession. I am happiest when the programs that I have written (YACC, Lint, the Portable C Compiler) are useful to others. In this regard, the contribution YACC made to the spread of Unix and C is what I'm proudest of.

Where do you see computer programming languages heading in the future, particularly in the next 5 to 20 years?

I like constraint languages, particularly because I think they can easily adapt to parallel hardware.

Do you have any advice for up-and-coming programmers?

You can't rewrite a program too many times, especially if you make sure it gets smaller and faster each time. I've seen over and over that if something gets an order of magnitude faster, it

becomes qualitatively different. And if it is two orders of magnitude faster, it becomes amazing. Consider what Google would be like if queries took 25 seconds to be answered.

One more piece of advice: take a theoretician to lunch.