FEATURE

Yacc, Unix, and advice from Bell Labs alumni Stephen Johnson

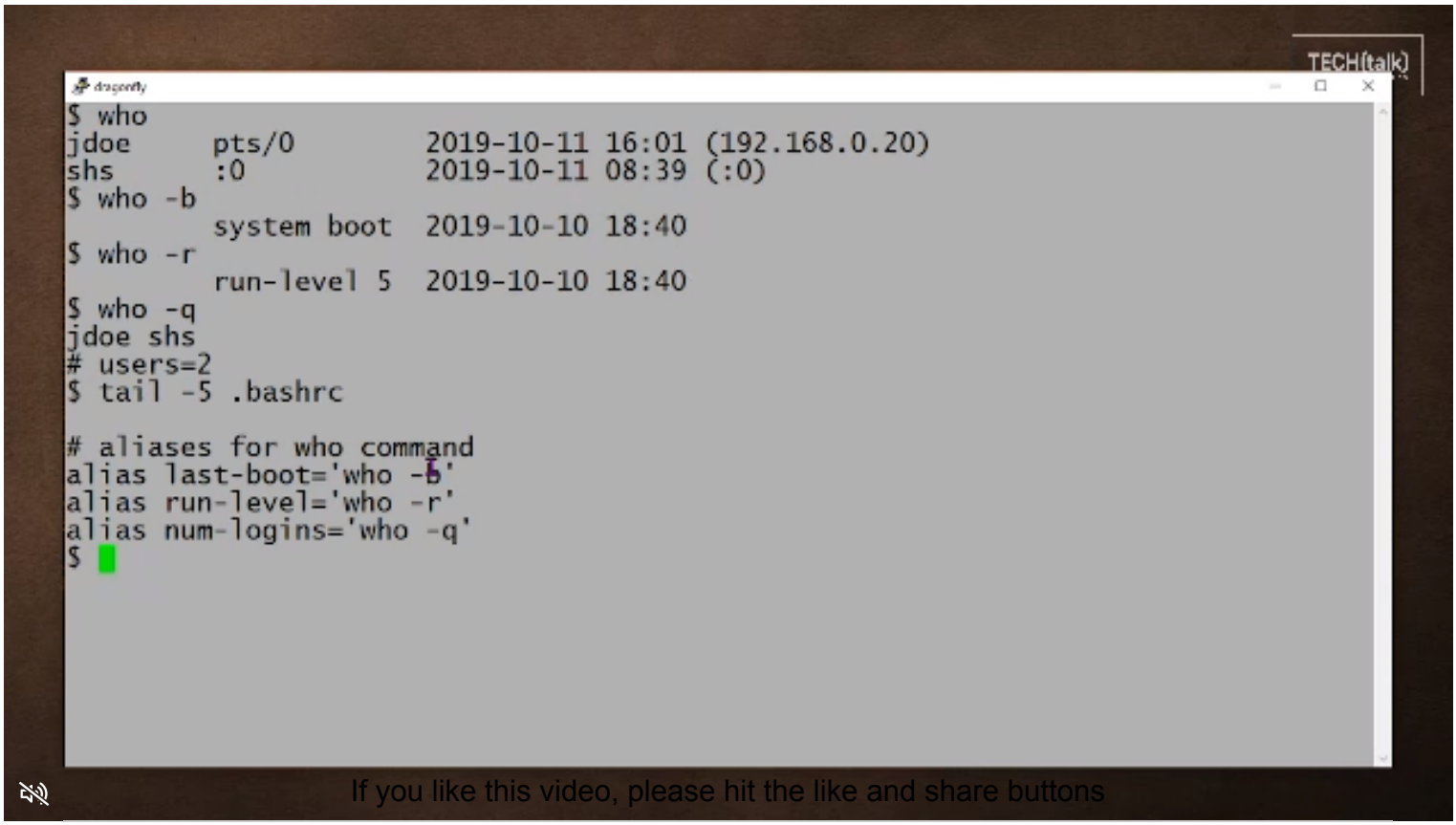
By Naomi Hamilton

Computerworld Australia |
JUL 10, 2008 12:00 AM PST

July 9, 2008 (Computerworld Australia) SYDNEY - Computerworld interviewed AT&T alumni Stephen C. Johnson about the development of Yet Another Compiler Compiler (YACC), part of a series of investigations into the most widely-used programming languages.

Previous interviews in this series include Alfred v. Aho of AWK fame, S. Tucker Taft on the Ada 1995 and 2005 revisions, Microsoft about its server-side script engine ASP, Chet Ramey about his experience maintaining Bash, Bjarne Stroustrup of C++ fame, to < a target="new" http://www.computerworld.com.au/index.php/id;766897508">Charles H. Moore about the design and development of Forth and a chat with the irreverent Don Woods about the development and uses of INTERCAL Johnson is employed at The MathWorks.

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 that 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!

[Related: Tech event calendar: Upcoming shows, conferences and IT expos]

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...

This story, "Yacc, Unix, and advice from Bell Labs alumni Stephen Johnson" was originally published by Computerworld Australia.

Follow everything from Computerworld

9 steps to lock down corporate browsers

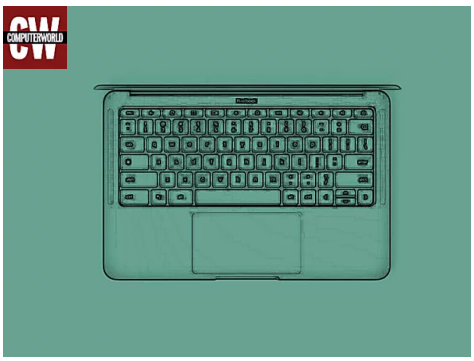
How Giffgaff is setting itself up for the 5G era
Homepage

Microsoft preps service pack-esque Windows 10 1909 for release
Homepage

Insurer LV= turns to Salesforce to speed up customer responsiveness
Homepage

This New $89 Trap Finally Solves The Japan Mosquito Problem
Moskinator

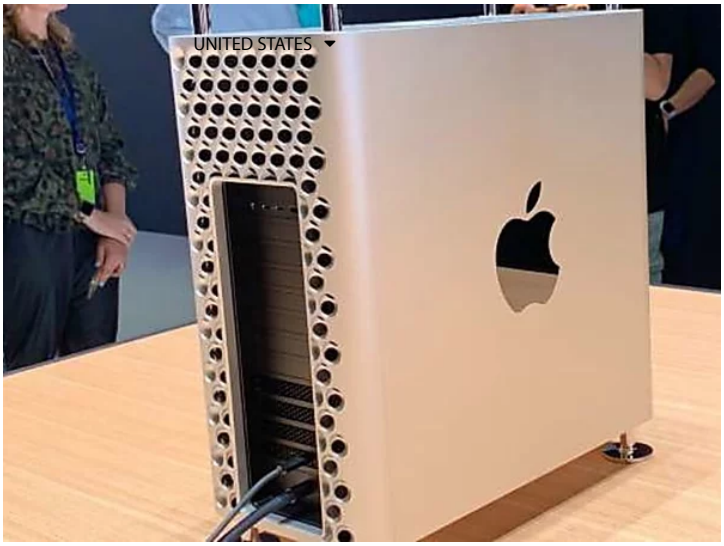
The Must-Play Fantasy City Building Game this Year
Elvenar



Making sense of Google's Pixelbook strategy
Homepage

Slack rolls out new Salesforce integrations, launches Workflow Builder
Homepage

Throwback Thursday: Hardware isn't his thing
Homepage



SHOP TECH PRODUCTS AT AMAZON