Avraham “Abe” Bernstein | Master S/W Engineer | CV

Represented by Tk Open Systems Ltd.

Version: 0.5.3-tkos

Last update: 2017-09-25T12:51:36Z

Secure[1] photo of the author, Avraham Bernstein c. 2010.

0.1 Contact Info & Links

email/skype: Avraham.Bernstein+tkos@gmail.com

geolocation: Jerusalem Israel, **tz:** UTC +02:00/+03:00 [winter/summer]

tel-IL-mobile/whatsapp: +972.54.641-0955

tel-US-mobile: +1.845.402-0023

www-home: http://purl.org/Avraham.Bernstein

linkedin: https://www.linkedin.com/in/AvrahamBernstein

cv-full: HTML, DOCX, PDF **[this file]**

cv-abbrev: HTML, DOCX, PDF

---

1 This secure photo was built using an oilify filter and a visible email address watermark. It uses digital camouflage that will defeat almost all face matching algorithms - in spite of the fact that my 3 year old grandson has no problem identifying me from this photo. It protects my privacy in two ways. (1) Harvesting it from the Internet, and adding it to a database of photos that will be compared with CCTV surveillance photos will not work. (2) The photo has built-in two factor authentication (2FA). The prominent email address watermark prevents a web site from maliciously displaying my photo while attempting to associate/label it with any other name besides mine. Similarly if a site were to maliciously display my photo without my permission in order to imply my association or agreement with them, then it is trivial for anyone who may be suspicious of their claim to “challenge” the site by asking me for a verification email.

0.2 Table of Contents

Avraham “Abe” Bernstein | Master S/W Engineer | CV....1

0.1 Contact Info & Links....1

0.2 Table of Contents....2

1.0 Summary....4

2.0 Work Experience....4

2017-present: Cybersecurity Consultant....4

2011-17: Security Policy Mngr & Architect: Cybersecurity: OTT Internet Pay TV System....5

2016-16: Cybersecurity Consultant: Protection of a Small Business with Extremely High Security Concerns....7

2010-11: S/W Architect & Developer: Transportation: Urban Traffic Vehicle Route Guidance Algorithms....7

2009-09: S/W Architect & Developer: Bioinformatics: PCR Algorithm....8

2004-09: Cybersecurity Researcher for a CA Satellite Pay TV System....8

2002-03: S/W Architect & Developer: Accessibility: Enabled Blind to “See” Maps....9

1999-2002: S/W Architect & Developer: Network: Utilities for a “Wireless” Cable Modem and Router System....10

2001-01: S/W Architect & Developer: Network: Network Management System (NMS) for a FSO Device....11

2001-01: Consultant: Network Management System (NMS) for a Cable Modem & Gateway System....11

2000-01: S/W Developer: Communications: Win32 Asynchronous TCP/IP DLL for a Visual Basic Project....11

1998-99: S/W Architect & Developer: Compiler: GCC Compiler Port for a 128-Core Stack Machine....12

1997-98: S/W Architect & Developer: Factory Automation: Conoscopic Interferometer Workstation....12

1996-97: Lecturer: Win32 Internals Course....12

1996-96: Consulting S/W Engineer: Win32 Improve Performance of a Soft Real-Time Biofeedback Application....13

1996-96: Consulting S/W Engineer: Win32 Device Driver for a Frame Grabber....13

1995-96: S/W Architect & Developer: US DOD Mil-Spec Automated Testing: Night Hawk Fire Control System....13

1995-95: Lecturer: Introductory University Computer Science Course on Database Theory....14

1991-94: S/W Architect & Developer: VLSI: Simulator & S/W Toolchain For DSPG PINE CPU....14

1989-91: S/W Architect & Developer: Factory Automation: Shop Floor Production Control (SFPC) System: BARI II....15

1988-88: S/W Architect & Developer: Accessibility: Quadriplegic PC Accessibility....15

1987-88: S/W Developer & VAX/VMS Sysadmin: 3D Printer: Solider....16

1986-87: S/W Developer: Soft Real-Time RS232 Z80 Communication Driver: Data Collection & Access Control Terminal....16

1985-86: S/W Developer: Factory Automation: Leather Sewing Workstation....16

1984-85: S/W Developer & VAX/VMS Sysadmin: Hebrew/English Word Processor: Glyph....17

1983-84: S/W Developer: Real-Time: Data Collection Terminal & Lavi Fighter Plane Radar....17

1981-83: S/W Developer & IBM CP/CMS Assistant Sysadmin....17

1979-80: Programmer & Economist....17

1977-78: Intervenor/Economist....18

3.0 Education....18

3.1 Formal Education....18

1979: York University, Canada: MA Economics & Applied Mathematics....18

1977: University of Toronto - Rotman School of Management (MBA Program): No Degree....18

1976: University of Toronto: BA Economics & Applied Mathematics....18

3.2 Continuing Education....19

4.0 Spoken Languages....19

5.0 Computer Languages, SDKs, and Operating Systems....19

6.0 Patents Under Development....20

7.0 Personal....20

Appendix A: Programming Language Preferences and Musings....20

Appendix B: Domain Specific Languages (DSL)....21

B.1: DSL What and Why.....21
B.2: DSL How To.....21
B.2.1: DSL Simple via Jinja2.....21
B.2.2: DSL Full-fledged.....23
Appendix C: How To Write Correct, Maintainable, Secure, and Easy-to-Test Code.....24
Colophon.....24

1.0 Summary

I am an experienced computer scientist and S/W architect. I have devised innovative solutions to many S/W problems for a wide range of fields, including

- cybersecurity
- cryptography
- bioinformatics
- factory automation
- VLSI CPU design
- telecommunications
- blind vision
- accessibility
- transportation vehicle route guidance
- automated testing

I have worked for a number of organizations, large and small, and helped them realize improvements in their product performance, often putting them in the front rank in their field. I have acquired expert knowledge in a number of fields, often liaising with noted experts, and have been able to quickly apply this knowledge to improve the competitive position of the companies and their products. I have a keen interest in computer languages, both practical and theoretical. I have created a number of domain specific languages (DSL) that were instrumental in greatly simplifying seemingly intractable problems.

In order to understand how I design S/W, see the following appendices:

- Programming Language Preferences and Musings
- Domain Specific Languages
- How To Write Correct, Maintainable, Secure, and Easy-to-Test Code

2.0 Work Experience

2017-present: Cybersecurity Consultant

@Self-Employed, Jerusalem:

Keys: cybersecurity, architect, algorithms, obfuscation, compiler, **C/C++**, javascript, WASM

1. I am developing an obfuscating compiler for C/C++ and for Web Assembly (WASM). Still in stealth mode.
2. I am a security mentor for the Jerusalem Mass Challenge start-up hub.

2011-17: Security Policy Mngr & Architect: Cybersecurity: OTT Internet Pay TV System

@Viaccess-Orca, Ra'anana - a subsidiary of @Orange France, and @Discretix/SansaSecurity, Netanya recently acquired by @ARM:

Keys: cybersecurity, DRM, architect, algorithms, anti-reverse engineering, obfuscation, LLVM compiler, cryptography, **C/C++**, TCL, Python, bash, Android root detection, Linux, ELF edit, IOS

1. The product was an Over-The-Top (OTT) Internet pay TV system. We provided the S/W infrastructure to our customers, the legacy (i.e. satellite and cable) pay TV operators, so they could also provide an OTT service to their subscribers in order that they could try to compete with Netflix. The system was designed for small screen Android and IOS devices, i.e. up to 10 inches. We used DRM to encrypt the content. Originally the DRM was Microsoft's PlayReady, and later on their own proprietary in-house DRM, and finally also Google's Widevine.
2. I was responsible for security policy and architecture. I worked closely with the product management and the S/W development team leader in order to determine security requirements, their costs and benefits, and was the architect of their implementation. In many cases the security features were very complex, so I first needed to create a working proof-of-concept, before finalizing their specifications.
3. Originally the exclusive focus of security was protecting the devices from leaking content and keys, i.e. from being reverse engineered. We relied heavily on the premise that we refused to play on "rooted" Android devices or "jail-broken" IOS devices.
4. But as time went by, rooted Android devices became inexpensive and ubiquitous in the consumer market. Therefore due to declining royalties, the major studios (e.g. Disney, Sony, Warner Bros., etc.) were economically forced to allow playback on rooted devices. Therefore additional security had to implemented on the back-end web servers, e.g. to check whether or not a subscriber downloaded an unusually high number of hours of content, or whether the subscriber had simultaneous downloads from different IP addresses. I designed a secure data logging system in order to better understand how subscribers were using the system, and in order to detect piracy. Given that we had tens of millions of subscribers, the economic challenge was to minimize communication costs of the logs.
5. I specified the anti-reverse engineering and obfuscation programming frameworks and libraries in C/C++.
6. My typical development methodology was to first build a prototype for desktop Linux, secondly as a standalone CLI application on the target device, and finally to hand over a working prototype to the development team. Whenever possible I preferred to test on virtual machines.
7. The challenges of implementing obfuscation are that (1) the other programmers should not be concerned about it because their focus must be on writing correct code, and (2) the resulting increase in size and reduction in run-time speed must not noticeably reduce the usability/functionality of the application. In general the aim of obfuscation is to provide "good enough security" that will deter 95% of

potential attackers, and when combined with regular application updates will force an attacker to begin his next reverse engineering attempt from scratch.

8. All secure code modules on the device were implemented as native libraries written in C/C++. Usually offline utilites were implemented in Python.
9. I developed a post processor to obfuscate the resulting binary object ELF files.
10. I developed a light weight obfuscated cryptographic library implemented as a H file using `inline` functions so that every module that included it had its own private copy of the library with a module specific randomized implementation which prevented an attack against a single core cryptographic module that could potentially subvert the whole application. During movie playback which relied upon AES decryption, for performance reasons we could not afford to also use AES encryption so we used light weight techniques instead, e.g. Xorshift PRNG stream cipher.
11. I developed an Android root detection mechanism using fuzzy logic techniques.
12. I developed a background watchdog security thread to dynamically ensure that the binary code had not been tampered with.
13. I developed vector operations for the C preprocessor that allowed a stream cipher to be applied to a constant string *pre-compile* time that was used to shroud function name strings that were dynamically loaded using `dlsym()`.
14. I developed a light weight method to efficiently shroud all system calls so that their address is calculated just-in-time before the call is made. It foils the professional reverse engineering debuggers Hex-Rays and OllyDbg which normally can automatically identify and place anchors on the system calls.
15. I created a prototype of a dynamic shared library (DSO) that formally exported no symbols. In fact it used an asynchronous back channel that allowed the DSO to communicate with its caller by using a function declared with the gcc constructor attribute that executes before `dlopen()` returns.
16. I was responsible for the purchase decisions and usage policy of 3rd party obfuscation and cryptographic utilities and libraries. The two main 3rd party utilities that we used were the InterTrust WhiteCryption SCP obfuscating C/C++ compiler and their SKB “whitebox” cryptographic library. I worked with the WhiteCryption team to specify new features for their tools. For example, even though their UPX-like tool statically encrypts and packs a DSO, it automatically decrypts it when it is loaded - allowing anyone to load it including an attacker who wants to do reconnaissance, so we added a password protection mechanism using a global operating system object, i.e. an environment variable or a file.
17. One of the most difficult architectural problems was that it was very difficult to patch/update the application in a timely manner once a subscriber had already downloaded it. The final application was created by our customers, i.e. the satellite/cable operators. We supplied them with core libraries only. Therefore we did not have access to the enterprise keys from the Google Play Store or Apple App Store that were required to automatically push an update, e.g. to modify a configuration file with a new password due to a security breach on a server. The original security architect naively assumed that we would achieve higher security by building a customer specific version of our core module where we “baked” (i.e. embedded) in the customer’s keys instead of using a separate customer specific configuration file to be dynamically loaded upon each invocation of the application. The customers were justifiably “paranoid” about demanding a very high level of quality assurance before making a release, because a single major bug could swamp them with tens of thousands of support calls where the cost of handling a single call could be greater than the monthly subscription fee. Typically

their release cycle was every 6-12 months. Therefore the customers would never dare allow us to automatically push code to their subscribers that they did not yet vet. Eventually I architected a technical solution that allowed us to automatically push configuration files to the subscribers where we would give the customers 7 days prior notice. The solution also required that the servers support two sets of passwords and/or subscriber credentials during a key transition period.

18. **At the end of my 6 year tenure there were 40M subscribers, and no security breaches.**

2016-16: Cybersecurity Consultant: Protection of a Small Business with Extremely High Security Concerns

@Anonymous, Jerusalem:

Keys: cybersecurity, privacy, anonymity, WordPress, static web site, Cloudflare, Windows, Android, Google Docs, Google Drive

1. The client publishes a web site that “outs” (i.e. exposes) terrorists.
2. Their WordPress web site was regularly attacked. And for reasons of personal physical safety, their management and researchers needed to remain anonymous.
3. Their enemies were well funded NGOs and government sponsored organizations.
4. Their original web site developers were not cybersecurity aware, and their researchers who often worked remotely (i.e. outside of the office) were typical computer users who were naive about cybersecurity and how to protect their anonymity.
5. After doing an initial risk assessment, my task was to implement graduated policies that improved their security without causing an upheaval to the way they worked.

2010-11: S/W Architect & Developer: Transportation: Urban Traffic Vehicle Route Guidance Algorithms

@TeleQuest (defunct), Jerusalem:

Keys: urban vehicle route guidance, architect, algorithms, Java, AWS

1. I designed and implemented algorithms along with a computational infrastructure for urban traffic vehicle route guidance similar to what Waze does today.
2. I coded in Java. I implemented the simulation infrastructure on the cloud on AWS.
3. The simulations showed that with a guided vehicle penetration rate of just 15%, a very accurate representation of traffic flow could be achieved.
4. Also the simulations showed that opportunistic guidance algorithms created secondary traffic jams after the penetration rate exceeded 20%. And when the penetration rate exceeded 40%, opportunistic algorithms generated more traffic congestion than using no guidance at all.
5. Therefore an optimal algorithm had to anticipate traffic flows according to time of day (along with other factors), and to allocate vehicles with similar sources and destinations to different routes in order to avoid secondary traffic jams.
6. The Java implementation challenge was cost efficiency because the simulations used huge amounts of RAM and ran continuously for days on expensive servers. I could not rely on the efficacy of the built-in memory garbage collector (GC), so I had to explicitly coerce memory allocations and frees.

2009-09: S/W Architect & Developer: Bioinformatics: PCR Algorithm

@Syntezza Molecular Detection (defunct), Jerusalem:

Keys: bioinformatics, PCR, algorithms, architect, mathematical programming, **C**, Python

1. The client's product was a PCR (= DNA amplification technology) kit for detecting MRSA (= an often lethal staphylococcus bacteria that is antibiotic resistant and thrives in hospitals) .
2. Within 3 months of joining the company and quickly learning the basics of bioinformatics via mentoring from a world renowned expert, Dr. Tzachi Bar, I discovered a new algorithm for overcoming PCR inhibition using Artificial Intelligence (AI) and Data Science techniques where too much sample "noise" prevents the result of the biological assay from being estimated by the classic sigmoidal shaped logistic function.
3. I implemented the final algorithm in C, while I built the prototype in Python using the Numpy, Scipy, and Matplotlib packages.
4. **When I started the project, the client's kit had only a 50% detection rate due to inhibition problems associated with their *preliminary* chemistry that could not separate the DNA from the mucus in the nasal samples. This result was grossly unacceptable for any medical test. The investors had lost confidence, and were about to pull out. I improved the test's accuracy to 95%, which was 10% better than their competitors from the pharmaceutical giants. I saved the client from liquidation.**
5. I am in the process of patenting my algorithm.

2004-09: Cybersecurity Researcher for a CA Satellite Pay TV System

@Cisco-NDS, Jerusalem:

Keys: cybersecurity, DRM, algorithms, cryptography, anti-reverse engineering, obfuscation, LLVM compiler, VM, QEMU, RPC, automated testing, S/W quality, **C/C++**, TCL, Python, Linux, bash, Win32

1. I worked on a wide variety of security related projects. My background task was to do C/C++ code security reviews. Typically secure coding is achieved by adhering to best programming practices.
2. I was a member of the architecture team for their in-house LLVM obfuscating C/C++ compiler.
3. I developed techniques using Virtual Machine (VM) technology to crack Digital Rights Management (DRM) schemes, and to subvert the random number generators which are the core initialization process for all cryptographic algorithms. I implemented these techniques by hacking the open source QEMU VM emulator written in C.
4. I wrote the technical specification for CCTV (Chinese government TV) to secure the TV feed of the 2008 Beijing Olympic broadcasts against international piracy. We almost won the contract, except Microsoft offered to do it for free.
5. I architected and implemented a hybrid simulator/emulator debugger written in C for legacy Set-Top Boxes (STB) that originally could be debugged only with printf statements to log files. My new debugger allowed source code on the PC to be

debugged using the MS Visual Studio IDE debugger while still viewing the results on the STB. Implementation was accomplished by reverse engineering of the STB middleware API. 80% of the middleware ran natively on the PC, while the STB low level H/W specific portions were implemented via an agent on the STB that was accessed via API calls that were implemented as Remote Procedure Calls (RPC).

6. I wrote an automated testing system in TCL and C/C++ for a satellite content delivery system for huge content, e.g. delivering ultra high definition movies to cinemas, and print newspapers for remote publishing. I created a Domain Specific Language (DSL) in order to execute the satellite operations. After studying the Win32 C/C++ source code of the satellite ground control station, I detected a major conceptual flaw which the architect refused to believe (because testers are not supposed to understand Win32 internals!). So I wrote a progressive test that brought the satellite to its knees at only 25% of its rated capacity. Afterwards the development team used my tool to develop their own unit test scripts, and to execute a system sanity test before checking-in any changes to the source control system.
7. I did a study for senior management by data mining the company's bug database, which showed them that 25% of S/W development manpower was wasted on fixing bugs. And I presented them with simple techniques that could reduce this number by 80%.

2002-03: S/W Architect & Developer: Accessibility: Enabled Blind to "See" Maps

@Virtouch (defunct), Jerusalem:

Keys: accessibility, blind, architect, algorithms, GIS, MapML, HTML, SVG, javascript, XSLT, XML Schema, XSLT, **C**, TCL

1. I was the inventor and architect of a system that allowed the blind to to "see" geographic maps and digital images that were prepared using industry standard GIS map descriptions such as MapML.
2. Maps were displayed on a standard HTML browser using HTML, SVG, javascript, XSLT, combined with audio feedback.
3. A touch tablet and stylus were used instead of a mouse because the blind can use a touch tablet much more effectively to navigate the screen compared to a mouse. The blind have their own sense of hand-eye coordination, and intuitively understand the stylus position on the tablet.
4. The core design principles were the following:
    - When the mouse/stylus hovers over a pixel or clicks on a pixel (depending upon its mode of operations), the pixel will be conveyed in 3 different sonic dimensions.
    - The x and y axis are divided into a uniform grid of 72x72 corresponding to a 6 octave 12 note chromatic scale per axis.
    - The x axis is associated with the first instrument say a piano, while the y axis is associated with the second instrument say a flute, so pixel position is conveyed via playback of 2 simultaneous notes.
    - Normally volume is associated with the 8-bit gray value of the pixel.
    - Reduce clutter by allowing the user to dynamically select which map layers to view. Typically visual maps display multiple layers simultaneously. Usually blind users cannot handle viewing multiple layers at once.

- Allow the user to dynamically select the level of feature granularity that he wants to view. Think of a topographical map that has an isoline granularity of 100 meters. A blind user may be more comfortable changing the isoline granularity to 500 meters.
- Depending upon the mode of operation, the rectangular touch tablet can be mapped into the whole rectangular screen, or into a rectangular window on the screen. The blind very quickly intuitively understand this type of mapping.
- Take advantage of SVG's seamless zooming capability.

5. Maps and images were prepared offline using XML Schema along with TCL preprocessing.
6. Note that this application required no custom H/W unlike their expensive (~$300) VTPlayer tactile mouse.
7. **I *almost* saved the company from liquidation. My research was awarded a European FP6 grant of $0.5M Euro that required *matching funds*. But the investors refused to put up the matching funds due to the company's long history of financial failure in the children's Braille education market using their expensive tactile mouse.**

1999-2002: S/W Architect & Developer: Network: Utilities for a "Wireless" Cable Modem and Router System

@Vyyo (defunct), Jerusalem:

Keys: network, architect, algorithms, SNMP, SNMP-agent, NMS, automated testing, **C**, TCL, embedded

1. I was the architect and designer of the SNMP network management system (NMS), MIB, and embedded SNMP agent. The workstation NMS was implemented in TCL, and the embedded SNMP agents on the cable modem and router were implemented in C.
2. I was the architect and designer of a hybrid IP connection for cable modems where there was no physical cable upstream channel. Instead the upstream channel used a telephone modem (ATA), while the downstream channel used the cable modem. Head end network equipment for both interfaces was supplied by the cable operator. My solution was to dynamically modify the arp table of the edge router. For typical surfing, the effective downstream rate was as fast as a pure cable solution. The server implementation was in TCL, and the embedded cable modem implementation was in C. The company applied for a provisional patent.
3. I greatly improved the efficiency of the laboratory modem speed stress testing by a factor of 10-100 by using a steepest descent search algorithm instead of a binary search algorithm. Reduced testing time per modem from hours to minutes.
4. I designed a virtual testing lab with 64K modems and 512K PCs via multiplexing the physical connections. The test lab had only 256 physical cable modems, 4 physical PCs with 8 network connections each, 1 cable router, and 2 24-port layer-2 programmable switches. By dynamically editing the PC MAC addresses, and by dynamically editing the MAC filters on the network switches, I was able to fool the router into believing that it faced 64K modems, and to fool the modems into believing that each one was shared by 16 PCs. The Linux workstation implementation was in TCL and C.

5. I designed a very efficient hash table algorithm in C for the router's arp table cache, based upon an algorithm I had invented 10 years earlier. The special features of the hash table algorithm were no use of dynamic memory allocations for embedded safety, a unique 2^N table size algorithm that required no use of division or modulo operations for efficiency, and a LIFO queue in order to gracefully handle table overflow.
6. I designed a flash memory file system for the modem and router in C.
7. I designed a TLV configuration file utility both offline for the PC workstation, and embedded on the modem and router in C and TCL.

2001-01: S/W Architect & Developer: Network: Network Management System (NMS) for a FSO Device

@MRV-Jolt (defunct), Jerusalem:

Keys: network, architect, SNMP, SNMP-agent, NMS, Java, **C**, TCL

1. The client's product was a free space optics (FSO) repeater that enabled fiber optic cables to be extended through the air via lasers for distances of up to 5 km.
2. These devices were made of digital H/W with no need for a CPU.
3. In order to provide them with SNMP network management capability, I selected an inexpensive micro-controller that could interface with the FSO H/W, had an Ethernet port, and a built-in Java interpreter.
4. I designed the MIB, and an Java program that could interface with the H/W.
5. I implemented a simple HTTP server on the board that acted as an SNMP proxy agent.
6. I wrote a CLI utility in TCL and C using the open source Net-SNMP framework to illustrate their SNMP compatibility to their customers.

2001-01: Consultant: Network Management System (NMS) for a Cable Modem & Gateway System

@One Path Networks - Foxcom, Jerusalem:

Keys: SNMP, NMS

I performed a one week requirements study in order to select the most appropriate NMS infrastructure for their needs. I saved them over $200K compared to their original selection.

2000-01: S/W Developer: Communications: Win32 Asynchronous TCP/IP DLL for a Visual Basic Project

@Inex-Zamir, Jerusalem:

Keys: TCP/IP communications, **C**, Visual Basic, Win32, soft real-time

1. The client's product was a vehicle license plate reader for their own camera controlled parking systems, and as an OEM component that they sold to operators of camera controlled toll roads.
2. They developed their parking system in Visual Basic (VB) on Windows PCs.
3. They came to me because they needed asynchronous network communication capabilities, even though VB is inherently single threaded.

4. I wrote a Win32 DLL in C for the client, and helped them to modify their VB code so it could incorporate asynchronous communications.

1998-99: S/W Architect & Developer: Compiler: GCC Compiler Port for a 128-Core Stack Machine

@Fourfold Technologies (defunct), Jerusalem:

Keys: gcc C compiler, architect, algorithms, DSL, **C/C++**, FORTH, LISP, TCL

1. This was a very challenging gcc port because the architecture had no registers, or alternatively an infinite number of registers, while RAM access was highly unusual in order to accommodate the 128 cores.
2. The machine instruction set was FORTH-like, so it presented unusual optimization challenges, because it was so unlike the standard CISC and RISC instruction sets.
3. I noted early in the project that the method for emitting target object code was well suited to an object oriented design, even though at the time the gcc compiler collection only interfaced with C code. In order to lower the "impedance" of using C++, I created a static "wrapper" for the C++ classes that could interface to the C code that the gcc compiler development framework generated.
4. The final source code was extremely repetitive. Therefore I developed a preprocessor in TCL in order to automatically generate much of the code.
5. The resulting C compiler worked and produced efficient code.

1997-98: S/W Architect & Developer: Factory Automation: Conoscopic Interferometer Workstation

@Newport-Optimet:

Keys: measurement workstation, architect, algorithms, DSL, **C**, TCL, OpenGL, Win32, soft real-time

1. The client's product was a measurement workstation based upon their conoscopic probe.
2. The object to be measured was placed on a static platter, while the scanning probe could be moved in the XY axes above the platter. The probe measured Z distance.
3. The workstation also included a video camera.
4. Generally the probe would scan at its maximum speed, although some types of materials, especially reflective, required lower speed.
5. My task was to design the S/W architecture for a Windows NT PC to implement an *automated* workstation that could be used in a factory environment.
6. I created a domain specific language (DSL) implemented in TCL and C for controlling and configuring the workstation.
7. The most interesting and challenging aspect of the project was to give "life", i.e. semantic meaning, to the millions of raw XYZ data points that were measured.
8. I used OpenGL for rendering graphics. I created a toolkit and domain specific language for the 3D visualizations - unfortunately just *before* the initial release of VTK.

1996-97: Lecturer: Win32 Internals Course

@M.E.R., Jerusalem:

Keys: lecturer, Win32, **C**

1996-96: Consulting S/W Engineer: Win32 Improve Performance of a Soft Real-Time Biofeedback Application

@MindLife-UltraMind, Jerusalem:

Keys: Win32, soft real-time, **C**

Besides greatly improving the performance of customer's relaxation/meditation S/W application, I also developed my own meditation technique that uses simple S/W or even no S/W at all, and can easily be taught to most people within 5 minutes. Already in the 1960's brainwave entrainment experiments showed that the brainwaves of cats, for examples, could be controlled by a strobe light source as long as the strobe frequency was within the normal frequency range of the cat's brain. Now human meditation takes place when the brain is exhibiting theta waves which are in the range of 4-8 Hz. Note that this frequency range is the equivalent of a metronome operating at frequency of 240-480 beats per minute (bpm). Normally mechanical metronomes don't operate at speeds higher than 180 bpm, but today (i.e. 2017) there are many free digital metronome applications (on smartphones, tablets, and desktop PCs) that operate at much higher speeds, where 240-360 bpm is the "sweet spot" for meditation. My favorite application is GNU GTick. I use the following 3 settings:

- 240 bpm (= 4 Hz), 4/4 time
- 300 bpm (= 5 Hz), 5/4 time
- 360 bpm (= 6 Hz), 6/4 time

The application has a strong first beat option in case you lose synchronization, and it also has the option of a visual mode that synchronizes with the audio. By keeping time with the metronome app, your brainwave frequency synchronizes to the metronome frequency, which by definition puts you into a meditative state. And after a little bit of practice, many people can simply watch a clock with a second hand that makes a single discrete jump every second, and silently count to 4, 5, or 6 per jump, which is equivalent to frequencies of 4, 5, and 6 Hz respectively.

1996-96: Consulting S/W Engineer: Win32 Device Driver for a Frame Grabber

@Visionix-Cefar, Jerusalem:

Keys: Win32, soft real-time

1995-96: S/W Architect & Developer: US DOD Mil-Spec Automated Testing: Night Hawk Fire Control System

@Pitkha Outsourcing (defunct), Jerusalem for @Elbit-Elop, Rechovot:

Keys: automated testing, mil-spec, architect, DSL, **C/C++**, lex/yacc BASIC compiler, Win32, soft real-time

1. The client's project was the fire control system for the Night Hawk laser guided missiles.
2. My task was to create an architecture and implementation of the automated test procedures dictated by the US DOD. The test specification documentation was over 1 meter high.

3. Additionally there would be many *ad hoc* tests that would be required during the development process.
4. The test equipment included external voltage regulators, external heating and cooling equipment, vibrators, etc.
5. The workstation controlling the tests was a Windows NT PC.
6. Instead of creating a monolithic test program in C/C++, I created a BASIC-like domain specific language (DSL) with special C++ drivers for controlling the various pieces of H/W.
7. I implemented the BASIC compiler using lex/yacc and C++.
8. I made this critical architectural decision because the test specification manual was being regularly revised, plus I needed the flexibility to allow the *non-programmer* system engineers to write their own scripts without delving into the underlying C++ code. And I wanted to avoid having the client call me every time he needed to implement a minor/trivial change that he could easily learn to do himself.
9. The standard tests were presented via a UX which in fact emitted BASIC script.
10. The system worked as planned. Unsupervised tests ran successfully for up to 72 hours (over holidays).
11. Post mortem: The first version of TCL was released just around the time of this project, i.e. *pre-Internet*. Once I learned about TCL, I immediately realized it would have been the ideal platform for creating the test environment. Instead of me having to create my own BASIC-like language with control structures and variable handling, I could have relied upon TCL, and simply added custom primitives for the various pieces of H/W.

1995-95: Lecturer: Introductory University Computer Science Course on Database Theory

@Michlala College Bayit Vegan, Jerusalem:

Keys: lecturer, database, SQL

1991-94: S/W Architect & Developer: VLSI: Simulator & S/W Toolchain For DSPG PINE CPU

@Pitkha Outsourcing (defunct), Jerusalem for @DSP Group, Givat Shmuel:

Keys: VLSI simulator, S/W Development Toochain, architect, algorithms, DSL, **C/C++**, lex/yacc, assembly, Win32

1. I was the S/W architect of a **clock accurate** DSP CPU simulator along with a complete software development toolchain, i.e. a debugger, C compiler, assembler and linker. Note that the system was developed just *before* the GNU Compiler Collection framework reached maturity, i.e. v2.95.
2. **This system enabled working applications to be developed before the chip became physically available. It reduced application time-to-market by 6-12 months.**
3. The technological breakthrough was my design of a domain specific language (DSL) that described the CPU architecture **including the pipeline**. Implementation was in lex/yacc and C++.
4. The associated DSL compiler automatically generated the source code for the complete toolchain that enabled it be automatically rebuilt within an hour in the face of almost daily changes to the VLSI architecture - especially the pipeline.

5. **And after every change to the architecture, we ran the complete suite of VLSI verification regression tests 100-1000 times faster than the VHDL simulator.**

1989-91: S/W Architect & Developer: Factory Automation: Shop Floor Production Control (SFPC) System: BARI II

@Digital Equipment Corporation (DEC) (defunct), Herzliya for @Iscar, Tefen:

Keys: factory automation SFPC, architect, algorithms, DSL, Pascal, SQL, VAX/VMS

1. Iscar Matkash in Tefen IL is a fully automated factory that produces thousands of different cutting blades using a sintering process. The raw materials go through many stages of operations. In many cases after undergoing intermediate processing, the partially processed material can still be diverted to multiple final products - similar to stem cells. The factory contains hundreds of automated workstations, stands, stacks, guided vehicles, and conveyor belts. The product or intermediate product is placed on pallets. The pallets are moved from one stand on a workstation to a stand on another workstation, or temporarily to a storage stand or stack.
2. My task was to create a computer program that automatically operated/orchestrated the factory.
3. When my co-architect and I started this project, we had zero background in industrial engineering. We were supplied with a mentor who brought us up to speed.
4. Eventually after months of discussions we created an architecture that was a textbook object oriented taxonomy - a “factory object kingdom”. The top level object was a “production instruction”.
5. We defined the attributes and methods associated with each object.
6. We created a descriptive, i.e. *non-procedural*, domain specific language (DSL) that was designed to be user-friendly for the factory engineer.
7. I wrote the language manual.
8. We used the language to configure the factory. We created a UX which emitted CLI script. But major updates to the database were implemented via very large CLI scripts of thousands of lines.
9. We mapped the language to a relational database.
10. We coded the system in Pascal. Given the inherent object oriented (OO) nature of the architecture, C++ would have been the ideal implementation language, but the project manager refused because he was not familiar with any OO languages.
11. After 18 calendar months, and 6 man-years later, the factory ran perfectly!

1988-88: S/W Architect & Developer: Accessibility: Quadriplegic PC Accessibility

@Cubital (defunct), Herzliya - a charity project funded by the company and their CEO Itzhak Pomerantz in cooperation with the Beit Levinson Rehabilitation Hospital, and the IDF Rehabilitation Unit:

Keys: accessibility, Prolog, PC-DOS

1. First of all, it important to note that this project took place in 1988 when speech recognition technology was still in its infancy, and exorbitantly expensive.
2. The H/W used for this project was the following:

   a. A light pen, i.e. an obsolete pre-mouse point and click device that synchronizes with the trace signal of the CRT video display, outfitted with special light weight military optics that increased its effective range from 5 mm from the screen to 800 mm.
   b. A standard accessibility sip-and-puff switch.
3. The light pen was mounted on the user's head by using a sturdy woman's plastic hair head band, while the sip-and-puff straw replaced the button on the light pen.
4. The S/W that I developed overlaid a virtual keyboard on top of the screen.
5. In novice mode aiming the pen at a key caused it to illuminate, while clicking on it entered it into the system as a virtual key stroke.
6. The problem with novice mode was that the virtual keyboard obstructed more than half the screen.
7. In expert mode, the virtual keyboard was hidden, but when the pen was aimed at an individual key, it would pop-up and become illuminated.
8. **This system was used to enable Shulamit Gabbai, a former school teacher who became quadriplegic by contacting polio (due to a terrible malfunction in the *Or Akiva* drinking water supply which became mixed with sewage), to become a book editor for *Maariv*. She was able to type 30 characters per minute.**

1987-88: S/W Developer & VAX/VMS Sysadmin: 3D Printer: Solider

@Cubital (defunct), Herzliya:

Keys: 3D printing, **C**, sysadmin, VAX/VMS

1. Solider was one of the first 3D printers. It was the size of a shipping container, but it embodied the same principles used in modern 3D desktop printers.
2. Since a 3D printer prints one layer at a time, conceptually it is similar to the way a regular printer prints one page at a time, so I designed a VAX/VMS print driver for the 3D printer in C.

1986-87: S/W Developer: Soft Real-Time RS232 Z80 Communication Driver: Data Collection & Access Control Terminal

@Elde (defunct), Jerusalem:

Keys: data collection terminal, **C**, RS232, Z80, embedded, real-time

1985-86: S/W Developer: Factory Automation: Leather Sewing Workstation

@Orisol, Lod:

Keys: sewing workstation, DSL, algorithms, AutoCad, **C**, awk, PC-DOS

1. This is the first time I developed a Domain Specific Language (DSL) in order to implement the sewing machine control program.
2. Note that leather is a natural product. Therefore no two pieces of leather are identical, so an automated sewing program must be intelligent in order to accommodate the shape of each unique piece of leather. The workstation included a video camera in order to enable real-time feedback.

3. The sewing machine had a maximum speed of thousands of stitches per minute which produced very high torque, so it had to be carefully controlled to slow down gradually in order not to damage the motor. Non-linear sewing paths required slower sewing speeds, while sharp turns required nearly a complete stop before taking the turn.
4. I added *annotations* to the AutoCad description of the pattern. These annotations included needle up/down, stitching speed, minimum and maximum margin widths, and image detection algorithm instructions according to the color and pattern of the leather.
5. I compiled the pattern and annotations into a soft real time control program implemented in C. I implemented the compiler in awk.

1984-85: S/W Developer & VAX/VMS Sysadmin: Hebrew/English Word Processor: Glyph

@John Bryce, Jerusalem:

Keys: word processor, **C**, sysadmin, VAX/VMS

1983-84: S/W Developer: Real-Time: Data Collection Terminal & Lavi Fighter Plane Radar

@DSI (defunct), Givatayim for @Elta/IAI, Ashdod:

Keys: data collection terminal, PL/M, 8080, RTOS, fighter plane radar, Jovial, embedded, real-time

1. This was my first job upon making Aliya to Israel.
2. Before receiving my security clearance, I worked on the development of a data collection & access control terminal.
3. I wrote an RTOS kernel in **PL/M** and **assembly** for the **8080** CPU because at the time no off-the-shelf alternative was available. My mentor was Menachem Malkosh. It was a formative learning experience.
4. After receiving my clearance, I worked on the embedded radar S/W for the Lavi fighter plane in Jovial.

1981-83: S/W Developer & IBM CP/CMS Assistant Sysadmin

@Mitre Corp, McLean VA:

Keys: APL, PL/1, sysadmin, IBM CP/CMS

Most of my programming was in APL. The APL functional programming mathematical vector language is still a relevant paradigm to the present day.

1979-80: Programmer & Economist

@JWWA.com, an economic consulting firm in the Washington DC area:

Keys: electric utility economics, Fortran, IBM MVS

I configured computer simulations of electric power generating systems for the purpose of costing and pricing models that were used to present multiple scenarios at Public Utilities Commission (PUC) rate hearings. The simulation language was written in **Fortran**, and execution was on a IBM 370 mainframe remotely accessible via WYLBUR.

1977-78: Intervenor/Economist

@Ontario Energy Board (OEB), Toronto:

Keys: electric utility economics

1. I was an intervenor at the ECAP'77 costing and pricing hearings.
2. Like all citizens, I had legal standing because I paid an electric bill.
3. I took over the marginal cost pricing (= peak load or time-of-day pricing) proposal of the Ontario Hydro (i.e. at that time the name of Ontario's electric generation and transmission utility) economists who were forced to drop their case due to extreme political pressure from the metallurgy processing industry who at the time relied upon electric blast furnaces.
4. I actively participated in the hearing sessions for about 9 months. I filed submissions, gave expert testimony, and cross-examined opposing witnesses.
5. **I argued my position very well. At 22 years old, I was the first public interest intervenor in the history of the OEB to be awarded costs.**
6. **I published an op-ed in The Globe and Mail, i.e. at the time Canada's newspaper of record, explaining the economic and political issues surrounding the case.**

3.0 Education

3.1 Formal Education

1979: York University, Canada: MA Economics & Applied Mathematics

I passed my final examinations in economic theory before I even started the program, so in order to achieve my required course credits the faculty agreed to allow me to be a special student at the Univ. of Toronto graduate faculty of engineering, where I took the majority of my courses. My major project was a computer simulation in **Fortran** how to cost efficiently operate a hydro-electric dam. I was mentored by an economist from Ontario Hydro (i.e. at that time the name of Ontario's electric generation and transmission utility).

1977: University of Toronto - Rotman School of Management (MBA Program): No Degree

I "dropped out" of school in the middle of the year after taking an advanced micro-economic theory course which analyzed the Ontario Hydro marginal cost pricing submission to the Ontario Energy Board (OEB) above, in order to take advantage of the unusual opportunity to present Ontario Hydro's case for them, which they were forced to drop for political reason. The following year I was able to apply my course credits to an MA Economics program at York University above.

1976: University of Toronto: BA Economics & Applied Mathematics

The most memorable and still useful courses I took were in statistics, experimental design, game theory, advanced calculus, and microeconomics.

In 1971 at the age of 15, for a high school computer science course, I wrote a computer program to play a perfect game of 3D 4x4x4 tic-tac-toe in Fortran on an IBM 1130. The computer had 16 KB RAM, and was the size of a refrigerator. It was arguably my most formative learning experience from

which I received the computer programming “bug” which I carry with me to the present day.

3.2 Continuing Education

1. Today the field of computer science is changing so rapidly that one’s formal education has a half-life of less than 5 years. Therefore in order to maintain my state-of-the-art professional edge, I am involved in an intensive effort of continuing education.
2. From 1991-96, pre-Internet, I used to spend one afternoon per week reading at the Hebrew University Jerusalem (HUJI) computer science library.
3. Afterwards with the advent of the Internet, more up-to-date computer science topics were available on the Internet, so going to the library was no longer the most efficient way to keep updated.
4. Since 2005, I have maintained a subscription to the O’Reilly Safari on-line tech library.
5. My daily dose of tech news comes from Slashdot.
6. I regularly watch TedX and Talks At Google video seminars.
7. The most fascinating feature of TedX talks is to watch and learn how world class experts in a wide range of fields are able to distill their special area of knowledge to intelligent laymen in just 18 minutes. Whenever I make a presentation, I attempt to emulate the best TedX speakers. Also I attempt to write presentations which emulate this TedX Art of Innovation Top 10 Format.
8. I regularly read the tech sections of the Israeli business newspapers Globes and The Times of Israel.
9. I have eclectic interests. I regularly research new topics in depth.
10. My browser bookmarks are my most important professional store of my knowledge. I use Firefox because it has the best built-in bookmarking feature, because it uses tags/labels. I have a well honed tag taxonomy.
11. I am an expert generalist, and an autodidact polymath, i.e. a self-learner in new fields who achieves expertise quickly.

4.0 Spoken Languages

1. English (5/5)
2. Hebrew (4/5)
3. French (2/5)

5.0 Computer Languages, SDKs, and Operating Systems

Language knowledge in order of expertise, based upon my current frequency of usage:

1. C, TCL, bash + posix text utilities, e.g. awk, sed, etc.
2. C++, python, make, html5, css, markdown, pandoc, jinja2
3. flex, bison, llvm, javascript, java, yaml, json, go
4. forth, lisp, prolog, apl, fortran, opengl, svg, xml schema, relax ng, xslt, perl, C#

Note that I write compilers and Domain Specific Languages (DSL), so learning a new language takes me only a few days.

O/S knowledge in order of expertise, based upon my current frequency of usage:

1. Linux

2. Android
3. Win32
4. IOS

6.0 Patents Under Development

- Bioinformatics: (a) An extremely accurate and simple noise reduction and normalization algorithm to improve the accuracy of the standard PCR Ct calculation, and (b) an Artificial Intelligence (AI) methodology for measuring the quantity of DNA in a bioassay where inhibition makes it impossible to estimate the Ct because no underlying logistic function (= a flat “S” shaped curve) exists.

- Cryptography: A set of non-linear cryptographic primitives using Hamming weight-like data dependent permutations which overcomes the well known limitation of using Hamming weights because they have a binomial distribution.

7.0 Personal

I was born in Canada in 1956. I have lived in Jerusalem Israel since 1983. I am married with 4 children, 2B + 2G, plus many grandchildren. I take physical fitness seriously. Once upon a time I was a judoka, and a classical guitarist. I was an IDF reserve soldier for 15 years, where I served as a combat soldier in the infantry in the Jordan Valley. In spite of the fact that I joined the army when I was 32 years old (Hebrew: *Shlav Betnik*), functionally, but unofficially, I served in the capacity of deputy company commander (Hebrew: *Samech Mem Pe*) which provided me with the opportunity to achieve rich personal growth, and enabled me to learn important managerial and leadership skills.

Appendix A: Programming Language Preferences and Musings

Because I have a strong background in compilers, I am knowledgeable in many computer languages, and I can learn a new language at an expert level very quickly.

My “go to” language for low level programming is C while still taking advantage of modern programming paradigms (i.e. encapsulation, composition, and interface as opposed to inheritance). The advantage of using C is that it is the universal interface language.

Where projects demand it, I can write C++ well too, but I am much more aware than the average programmer of its potential pitfalls, for example:

- brittle and tightly coupled object hierarchies
- preference for inheritance when composition would suffice
- real-time limitations
- in general lack of WYSIWYG understandability of the source code due to use of complex inheritance hierarchies, operator overloading, cast operators, and exceptions.

However I find there is one class of application where I always prefer C++, namely when writing compilers and interpreters, because abstract syntax trees (AST) have a deep inheritance structure.

Appendix B: Domain Specific Languages (DSL)

B.1: DSL What and Why

First of all, here is the Wikipedia entry for domain specific languages (DSL).

Consider a GUI calculator app. It is not convenient to manually enter thousands of calculations. Similarly “fuzzing” the app, i.e. automatically testing it for corner conditions, will require that thousands of possibilities be tested. Most GUI apps are automatically tested by simulating low level UI events, e.g. mouse movement and button events, and individual key stroke events. These low level UI events have no semantic meaning *per se*. And if the UI layout were to change then a test program based upon UI events would have to redesigned. It would be much much easier to use this app for batch processing by starting off with all the data in a text file say in a simple CSV (spreadsheet) format. If the GUI calculator app were built to emit script acceptable to the venerable Linux `bc` CLI calculator app, then it would be trivial to automate it. And for a larger and more complex GUI app with lots of menus and dynamic dialog boxes, interpreting UI events is much more difficult.

Now consider the venerable Linux `gdb` debugger which has a built-in CLI. Most GUI debugger apps on Linux are in fact front ends for `gdb`. It is usually not too difficult to write a huge `gdb` script which can be used to fuzz individual functions inside a large executable program (as long as it is compiled with debug symbols).

In general an app’s GUI should be a wrapper for the underlying app’s CLI. The GUI should emit CLI script. This way it is easy to record user GUI events. The app can be used in batch mode. And it is straightforward to write an automated testing suite for the app.

When specifying an architecture for a large system, it is preferable to create a DSL that describes all aspects of how the system is supposed to behave. This DSL formalizes the architecture, and provides the groundwork for automated testing.

The app’s CLI should include end user definable control structures (i.e. condition and loop handling), along with the ability to define procedures, variables, and data structures. Embedded language frameworks such as FOSS Lua and FICL (i.e. an embeddable FORTH) provide this infrastructure for free. In fact these relatively small frameworks, i.e. with a memory footprint of 50-100 KB, usually grossly simplify the CLI design. Now an argument to a CLI method can take a numeric expression without the necessity of the programmer to understand anything at all about the complexity of writing a numeric expression parser. The framework allows an end user to create his own macros, test conditions, and write loops. An app with a CLI can be “glued” together with many other unrelated apps, just like the hundreds of Posix text utilities, and used in many unexpected but very useful ways - as opposed to a GUI-only interface which greatly limits how an app can interface with other apps.

B.2: DSL How To

B.2.1: DSL Simple via Jinja2

Most languages have mediocre or non-existent generic, macro, and template facilities but I can get around these limitations by wrapping most any language using the superb Jinja2 template/macro language as a preprocessor. Google uses Jinja2 in this way for its flagship Chromium project. My first step before designing a full-fledged DSL is to determine whether or not Jinja2 can be gently coerced to do a good enough job.

For example, consider a long repetitive declarative configuration file, that could be greatly simplified and shortened by using Jinja2 templates.

Or consider the C language built-in cpp macro preprocessor which is very simplistic. It has no template support. There is no *pre-compilation* support for arithmetic or string operations, array index selection, execution of O/S shell commands, and there is no straightforward way to implement loops, while complex macros with many levels of calls can often be incorrectly considered to be recursive which abort without even an error message. Once upon a time the m4 macro processor would be the tool of choice to implement complex macros. It has an arcane syntax, but it is admittedly Turing complete. Today very sophisticated Jinja2 macros and templates can be wrapped to look like cpp macros. This wrapping is important because most modern C/C++ IDEs have language sensitive editors that will choke when encountering invalid C/C++ syntax. Therefore the Makefile implementation strategy is to first execute cpp (i.e. `gcc -E`), followed by Jinja2, followed by gcc.

The following is an example of a Jinja2 macro `J2_ABC` that executes a python function that can be declared in a C/C++ file, and will keep the language sensitive IDE happy. Because some IDEs with smart language sensitive editors interpret cpp macros during an editing session, there must be 2 modes of operation, (1) build mode that will emit Jinja2/Python code, and (2) IDE mode that will emit default syntactically valid C/C++ code.

```
/* !!! JINJA2 SYNTAX GOTCHYAS !!!
================================================================
=========================
1. J2 statements and expressions should be wrapped in cpp macros,
   otherwise language sensitive editors found in many of the IDEs will choke on
   the source code. Therefore the cpp preprocessor must be run *before* the J2
   preprocessor.
2. However it can be an effective technique to include pure J2 source files by
   protecting them with an *external* cpp guard:
                #if J2_MODE == J2_MODE_BUILD
                #include "my-macros.j2"
                #endif2.
3. The J2 delimiters that we chose to use are the following:
        {'%' ... '%'}: control statement
        {'#' ... '#'}: comment statement
        {'^' ... '^'}: embedded expression
        '%%':          line control statement
        '%''#':        line comment statement
4. Cpp definitions must specify J2 delimiters *indirectly*,
   otherwise will cause J2 preprocessing errors. See lines 35-41.
5. Must *not* use the default J2 line comment '#' which conflicts with cpp.
6. Must *not* use J2 default expression delimiter, i.e. double braces,
   otherwise will conflict with C/C++ syntax.
================================================================
=========================
*/

#define J2_MODE_IDE             0
#define J2_MODE_BUILD             1
#ifndef J2_MODE
// default allows seamless editing even with a smart language sensitive editor
#define J2_MODE                 J2_MODE_IDE
#endif
```

```
#if J2_MODE == J2_MODE_BUILD

#define J2_COM_DELIM            #   // comment statement delimiter
#define J2_CNTRL_DELIM           %   // control statement delimiter
#define J2_EXPR_DELIM            ^   // embedded expression delimiter

#define J2_CNTRL(...)           {J2_CNTRL_DELIM __VA_ARGS__ J2_CNTRL_DELIM}
#define J2_COM(...)             {J2_COM_DELIM __VA_ARGS__ J2_COM_DELIM}
#define J2_EXPR_2(ide_dflt_in_parens,...)  {J2_EXPR_DELIM __VA_ARGS__ J2_EXPR_DELIM}

#else // J2_MODE_IDE

#define J2_IDENT(...)           __VA_ARGS__

#define J2_CNTRL(...)
#define J2_COM(...)
#define J2_EXPR_2(ide_dflt_in_parens,...)  J2_IDENT ide_dflt_in_parens

#endif // J2_MODE

#define J2_EXPR_EMPTY(...)       J2_EXPR_2((),   __VA_ARGS__)
#define J2_EXPR_ZERO(...)        J2_EXPR_2((0),  __VA_ARGS__)
#define J2_EXPR_ONE(...)         J2_EXPR_2((1),  __VA_ARGS__)
#define J2_EXPR_CHR(...)         J2_EXPR_2(('), __VA_ARGS__)
#define J2_EXPR_STR(...)         J2_EXPR_2((""), __VA_ARGS__)

#define J2_ABC(...)             J2_EXPR_ZERO(py_abc(__VA_ARGS__))
```

B.2.2: DSL Full-fledged

When writing a full-fledged DSL, unless run-time efficiency or a binary target is critically important, I tend to shy away from using formal compiler frameworks such as flex/bison, ANTLR, LLVM, etc., because they are complex and have a relatively steep learning curve.

In my experience most DSLs require a limited number of domain specific verbs and data structures. The rest of the language such as control structures (e.g. conditions and loops), and procedure, variable, and data structure definitions, which are by far the most difficult and time consuming to develop, can piggyback upon the built-in features of extensible languages such as TCL, Python, Lua, FORTH, etc.

Where the implementation will be a standalone CLI, my first choice is to use TCL. And if the TCL app also needs a UX then the Tcl/Tk widget toolkit can be used to simply and quickly produce a GUI - admittedly not the most elegant, but usually good enough. However when third party packages are needed to support the language (e.g. mathematical, scientific, and AI libraries etc.) then Python is required. Admittedly there are at least 100 times more Python programmers than TCL programmers, so even though it usually requires significantly less code to create a standalone CLI app in TCL, from the client's perspective of future maintainability, it is often more cost effective to spend the extra initial effort to build the app in Python.

And where the implementation will be embedded inside another application then Lua is the easiest to understand for most end users, but where minimizing code footprint and maximizing execution speed are more important then FORTH/FICL is required.

Appendix C: How To Write Correct, Maintainable, Secure, and Easy-to-Test Code

1. I design applications that can be dynamically configured and automatically tested via a CLI script.
    - Ideally the CLI definition should already be part of the architectural specification of an application domain specific language (DSL).
    - Ideally the CLI should be wrapped with an industry standard light weight embedded interpreter such as Lua or FORTH/FICL.
    - Tip: In debug mode *only*, the interpreter should be allowed to invoke O/S shell commands.
2. I design modules with built-in test points.
    - I write CLI scripts to access and to fuzz these test points.
    - This technique promotes simple regression testing.
3. My preferred UX design is to have the UX generate CLI script - as opposed to directly invoking internal functions.
    - This allows UX actions to be *semantically* captured as opposed to capturing low level UI events (e.g. individual mouse and keystroke events), and allows functional testing to be independent of the UX.
    - And it allows large test and configuration scenarios to be first sketched with the UX, and then to be extended with a text editor.
4. Especially in languages such as C/C++ that use manual memory management, using a tool such as Valgrind to discover memory leaks is critically important.
5. I liberally use both static and dynamic assertions in my source code which are especially important during initial development. I am a big fan of the design by contract paradigm.
6. I use lint, and I heavily make use of compiler attributes that enforce safe code.
7. I analyze my code with complexity metrics. See cyclomatic complexity (McCabe) and Halstead complexity.
8. Never release code when the build process generates compiler warnings.
    - I have worked on too many projects where the build process generates hundreds of thousands of compiler warnings. In such a situation it is nearly impossible to determine which warnings are serious, so in fact all warnings are ignored.
9. A necessary condition for secure code is that the code first must be correct. See the Google Tech-Talk The Lazy Programmer's Guide To Computer Security.
10. And after all of the above techniques have been incorporated into the implementation, still the most efficient way to flush out 50% of the bugs is an informal code review with a colleague where the programmer must explain every line of his code. Typically during his explanation, the programmer has many "aha" moments when recognizing many of his own bugs.

Colophon

- **Generator:** This document was generated using the Pandoc universal document converter extended Markdown engine, along with the Jinja2 macro/template preprocessor. See the source code at my github site.