28th EuroForth Conference

September 14-16, 2012

Exeter College
Oxford
England

Preface

EuroForth is an annual conference on the Forth programming language, stack machines, and related topics, and has been held since 1985. The 28th EuroForth finds us in Oxford for the second time (after 1997). The three previous EuroForths were held in Exeter, England (2009), in Hamburg, Germany (2010) and in Vienna, Austria (2011). Information on earlier conferences can be found at the EuroForth home page (`http://www.euroforth.org/`).

Since 1994, EuroForth has a refereed and a non-refereed track. This year there were two submissions to the refereed track, and one was accepted (50% acceptance rate). For more meaningful statistics, I include the numbers since 2006: 13 submissions, 8 accepts, 62% acceptance rate. Each paper was sent to at least three program committee members for review, and they all produced reviews. One paper was co-authored by a program committee member, who was not involved in the review; the reviews of all papers (including the one co-authored by the PC member) are anonymous to the authors. I thank the authors for their papers and the reviewers and program committee for their service.

Several papers were submitted to the non-refereed track in time to be included in the printed proceedings. These online proceedings (`http://www.euroforth.org/ef12/papers/`) also contain papers and presentations that were too late to be included in the printed proceedings. Also, some of the papers included in the printed proceedings were updated for these online proceedings.

Workshops and social events complement the program.

This year's EuroForth was organized by Janet Nelson.

Anton Ertl

Program committee

M. Anton Ertl, TU Wien (chair)
David Gregg, Trinity College Dublin
Ulrich Hoffmann, FH Wedel University of Applied Sciences
Jaanus Pöial, Estonian Information Technology College, Tallinn
Bradford Rodriguez, T-Recursive Technology
Bill Stoddart

Additional Reviewer

Reuben Thomas, Adsensus Ltd.

Contents

The N.I.G.E. Machine: an FPGA based micro-computer system for prototyping experimental scientific hardware

Andrew Read

May 2012

anding_eunding@yahoo.com

Abstract

This paper describes the N.I.G.E. Machine, a user-expandable micro-computer system that runs on an FPGA development board and is designed specifically for the rapid prototyping of experimental scientific hardware or other devices. The key components of the system include a stack-based softcore CPU optimized for embedded control, a FORTH software environment, and a flexible digital logic layer that interfaces the micro-computer components with the external environment. The system has been demonstrated on a Digilent Nexys 2 development board and in an example scientific experiment involving a light source and sensor.

1 Introduction

1.1 Overall concept

The N.I.G.E. Machine's primary intended application is as an electronic control and measurement unit for experimental scientific apparatus. The concept is to combine the merits of a traditional computer-based control system with the flexibility of Field Programmable Gate Arrays (FPGAs), and a rapid prototyping environment. This is done using a FORTH based, harmonized hardware-software system and a commodity FPGA development board.

FPGAs are integrated circuits with reconfigurable internal logic components and interconnects that can be repeatedly reprogrammed with fresh logic designs at the time of use. The functional capability of FPGAs is broadly equivalent to traditional integrated circuits, although operating parameters differ. Logic designs are written in a hardware description language such as VHDL or Verilog and then downloaded to the FPGA as a binary file after being synthesized by the development tools of the relevant FPGA manufacturer. Typically new logic designs are tested on special circuit boards (development boards) that host an FPGA alongside commonly used peripheral components such as external memory, switches, indicators, and connectors.

The N.I.G.E. Machine is a complete, user-expandable micro-computer system with extensive input/output (I/O) capabilities, hosted on a low-cost FPGA development board. It comprises (a) a general purpose, stack-based, 32-bit softcore CPU that has a number of optimizations for embedded control such as deterministic execution and rapid interrupt response time, (b) FORTH system software, and (c) a set of digital logic interface components including both peripherals for development use (keyboard, video) and also user-expandable peripherals for application specific interface logic (interrupt controller, hardware registers, and various I/O ports).

The softcore CPU and the other digital logic components are coded in VHDL[1] and the FORTH environment is coded in the assembly language of the softcore CPU. The N.I.G.E

Machine has been implemented and tested on a Digilent Nexys 2 development board with a Xilinx Spartan-3E XC3S1200E FPGA. With a keyboard and video monitor connected to the board, the system is a fully operational, stand-alone native FORTH environment from power on. A short video has been made to demonstrate[2].

1.2 Engineering approach

Typically there will be three layers of engineering in an application for prototyping an experimental scientific apparatus:

The physical layer: The scientific hardware, with actuators and sensors, driven by some electronic circuitry. The external electronic circuitry is connected to the N.I.G.E. Machine through the numerous expansion connectors available on the host FPGA development board, routing to free I/O pins on the FPGA.

The digital logic interface layer: Factory or custom (written in VHDL) digital logic interface peripherals directly interface with the external electronic circuitry and provide I/O functionality for the micro-computer system. The use of application-specific, custom digital logic interface peripherals permits greater scope for high-speed pre-processing of external signals and for processing multiple external signals in parallel than is possible with typical fixed-logic I/O ports. In addition, complete flexibility is permitted in the interface specification and design. The digital logic interface components are connected into the micro-computer architecture largely through hardware registers and external interrupts.

The micro-computer layer: The micro-computer system, built on the softcore CPU and the FORTH system software, is the platform both for the interactive prototyping of the external hardware and the custom digital logic interface, and also for running the final operating software.

1.3 Comparison to alternatives

Other platforms have existed for some time that also combine elements of this engineering approach, for example:

- The use of a digital logic layer interfacing with the scientific apparatus parallels that of the CMS and ATLAS detectors at CERN. In the case of these detectors, the volume of measurement data and the speed with which it is generated necessitates a digital logic layer ahead of the computer layer to identify potentially interesting events and filter out the remainder before further processing. The CERN detectors are very high performance designs and extremely expensive.
- Numerous commodity, easy-to-use microcontrollers, such as the Atmel ATMEGA or Parallax Propeller provide a CPU with direct access to digital I/O ports alongside a selection of fixed-function hardware resources such as counters and timers. FORTH is sometimes implemented on such microcontroller platforms.
- Standard commercial or open source FPGA softcores are available that can be configured directly by the FPGA development tools, for example the Xilinx MicroBlaze[15].
- A number of small softcores have been designed specifically to execute FORTH[3, 4, 5, 6, 7, 8]. Several aspects of the J1[3] have directly inspired this project.

The N.I.G.E. Machine does not intend to compete head-on with any one of these alternatives, but rather offer something novel in the way that it brings different aspects together to create a flexible platform particularly focused on enabling rapid, small-scale, scientific research and development. In particular:

- The N.I.G.E. Machine works with FPGA development boards that are easy-to-use and affordable, so they are within the range of small labs and individuals.
- The N.I.GE. Machine is a fully integrated micro-computer system with video and keyboard facilities (rather than a purely embedded system).
- No complex or slow-to-use tool chain is required for software development. FORTH is instantly available at power-on.
- Rather than than relying on generic, fixed-function hardware resources, the N.I.G.E. Machine's digital logic interface layer can be custom built and optimized for each new application.
- The N.I.G.E. Machine's stack-based softcore is a full-featured, general purpose CPU that includes functionality such as interrupts, flexible memory access, and debugging facilities.

1.4 Scope of this paper

The primary aim of this paper is to describe the new softcore CPU and this is the main focus of section 2, "Methods". The paper also aims to illustrate the novel hardware development platform offered to users of the N.I.G.E. Machine and the use of FORTH for the development and testing of experimental apparatus. These are the main focus of the example application described in section 4, "Application discussion".

2 Methods

2.1 Key design objectives

In order to meet the goals outlines above, key design objectives for the softcore CPU were set out under the headings of (1) platform, (2) real-time control, and (3) CPU performance. These objectives and the strategies devised to meet them are as follows.

2.1.1 Platform objectives

Rapid prototyping To shorten application and software development time for users it was decided to (a) not require the use of an external tool-chain for programming and (b) have an interpreter available at power on. An interpreter allows the user to experiment directly with the electronic apparatus and also test small routines for bottom-up software development. BASIC was considered, but FORTH was chosen because it provides both an interpreter and also a compiler which can produce executables that are almost as fast as directly assembled machine code. For a new computer system, FORTH also has the advantage that it can be implemented quite easily in assembly language.

Custom digital logic interface layer The scope for creating custom digital logic interfaces in VHDL for each new application differentiates the N.I.G.E. Machine from a conventional microcontroller platform with resident FORTH, and opens up new design possibilities. The system can offer this firstly because it is built with soft-logic in an FPGA and secondly because the key micro-controller interface components (principally the memory-mapped hardware register module and the prioritized hardware interrupt controller) have been made user-expandable.

Stand-alone usage Alongside the CPU softcore and FORTH, a full set of peripheral modules (e.g. keyboard, video, other I/O) have been developed to create a complete micro-computer system for stand-alone use.

2.1.2 Real-time control objectives

Fast interrupt response time Fast interrupt response time facilitates the high frequency, low-latency processing of external signals[9]. One of the key handicaps to interrupt processing is the need to save the state of a large register set. Sympathetic with the choice of FORTH for the system software, the softcore CPU is stack based and so no save/restore of registers is required for interrupt (or subroutine) processing[11]. The N.I.G.E. Machine's typical interrupt response time is only 2 cycles to branch to the interrupt vector table followed by 3 cycles to branch through the vector to the interrupt routine itself.

Deterministic execution Deterministic execution[9] is the certainty that a given set of instructions will execute in a given number of CPU clock cycles regardless of state. Conversely, without deterministic execution, jitter is the deviation of an expected periodic signal from true periodicity. Avoiding jitter in electronic interfaces is essential for precise control and measurement. The instruction set and CPU control unit have been designed so that all instructions execute in a fixed number of cycles, including conditional branches and static RAM (SRAM) memory access. The CPU's execution pipeline has been designed so that there are no conflict states that could result in missed cycles.

Fast branch performance Fast, deterministic, branch performance is important to optimize response times in an embedded application[9]. On the N.I.G.E. Machine conditional and unconditional branches (BEQ and BRA) are designed such that they always execute in only 3 clock cycles (2 cycles for decode, 1 cycle for memory).

Maximum code density The fastest memory resources available to a softcore CPU are FPGA SRAM blocks. These also have the advantage over external memory of deterministic access (i.e. guaranteed single clock cycle read/write). However FPGA SRAM resources are typically limited to a several tens or hundreds of kilobytes. To maximize the use of SRAM as program memory, code density needs to be as high as possible[9]. On the N.I.G.E Machine almost all instructions are encoded in a single byte. This is achieved by using microcode as opposed to a hardwired decoder in the CPU.

2.1.3 CPU performance objectives

High instruction throughput High instruction throughput translates directly into higher processing performance. Without super-scalar features or parallel cores, the best achievable goal is throughput of one cycle per instruction. The N.I.G.E. Machine's CPU design features a three-stage execution pipeline that delivers single cycle throughput for most instructions.

Flexible memory access The N.I.G.E. Machine is a 32-bit (longword) CPU and all system memory is byte addressable. To optimize the speed and flexibility of memory access: (1) separate CPU instructions have been created to read and write memory in byte, word, and longword format, (2) the CPU is designed with three separate memory buses (one for SRAM access and two for byte and word access to the external pseudo-static dynamic RAM (PSDRAM) that is part of the Nexys 2 development board), and (3) even address alignment is not required when accessing word or longword data in SRAM system memory.

Fast subroutine performance As an optimization for the execution of FORTH, the instruction set includes a compound RTS (return from subroutine) instruction that can be overlaid on top of most single byte instructions, saving one clock cycle on each subroutine return. This follows the design of the J1 processor[3].

2.2 Limitations

Hardware and design tradeoffs also resulted in some limitations of the N.I.G.E. Machine as set out below. None of the limitations are fundamental to the design and hopefully they will be addressed in future developments.

Program memory space The softcore CPU is currently only able to execute instruction code located within FPGA SRAM memory. The external PSDRAM on the Nexys 2 development board memory cannot be used as a program memory store unless its contents are first copied to SRAM for execution.

Narrow instruction fetch Instructions are fetched from SRAM one byte at a time. This means that instructions requiring several bytes (mainly the load literal instructions) necessarily take several cycles to execute.

Lack of floating point The current implementation of FORTH does not include floating point software routines nor does the digital logic design include any floating point functionality in FPGA hardware.

Blocking interrupts The interrupt scheduler provides interrupt prioritization but once an interrupt is in progress it will block all other interrupts, even those of higher priority.

Lack of double precision (64-bit) arithmetic Some FORTH words in the ANSI core set require double precision division. However the current implementation of FORTH does not include double precision arithmetic software routines. Additionally the hardware dividers used in the CPU are limited to 32-bit operands.

2.3 Implementation of the CPU and other digital logic

2.3.1 CPU instruction set

The softcore CPU has 63 instructions as follows:

Stack manipulation 15 instructions: NOP (no operation), the FORTH words DROP, DUP, ?DUP, SWAP, OVER, NIP, ROT, >R, R@, R>, plus four words for loading or saving the parameter and return stack pointers

Math operations 12 instructions: +, -, NEGATE, 1+, 1-, arithmetic shift left and right, signed and unsigned multiply, add and subtract with carry, and signed and unsigned divide

Comparison operations 11 instructions: the bitwise equality tests = and <>, signed comparisons <, >, unsigned comparisons U<, U>, comparisons with zero: 0=, 0<>, 0<, 0>, and FALSE, which returns zero

Bitwise operations 7 instructions: the Boolean operations AND, OR, INVERT, XOR, logical shift left and right, and byte and word sign extension to 32 bits

Memory operations 6 instructions: FETCH and STORE of byte, word, or longword values

Load literal operations 3 instructions: LOAD longword, word or byte values from within the program code

Flow control 6 instructions: JMP (jump to the address on the parameter stack), BSR and JSR (branch/jump to subroutine), RTS (return from subroutine), and BEQ, BRA (conditional and unconditional flow control)

Exception handling 3 instructions: TRAP (software trap vector) RTS_TRAP (a single-step), and RTI (return from interrupt)

2.3.2 Memory data format

Data is stored in memory in big-endian format. That is, for multi-byte data the highest value byte is stored at the lowest numbered memory address. By way of context, Motorola 68k processors also use big-endian format while Intel processors use little-endian format. Either format could have been implemented in the softcore CPU but the big-endian format was found to be more suitable for the design of the shift registers that fetch data from memory, as well as being the author's preference because of its Motorola 68k heritage.

2.3.3 Instruction set encoding

The default instruction size is a single byte, encoded as follows (figure 1): bit 7 identifies whether the instruction is a branch or an ordinary instruction. If the instruction is a branch then bit 6 specifies if the branch is conditional or unconditional. If the instruction is ordinary (not a branch) then bit 6 specifies whether a return from subroutine is to be taken along with the execution of the instruction (this is the compound RTS instruction). For ordinary instructions, bits 5 – 0 (figure 2) are read as an integer that identifies the instruction in question (i.e. the "opcode"). For branch instructions bits 5 – 0 of the instruction are read as the high part (bits 13 – 8) of the branch address, with a following byte holding the low part (bits 7 – 0) of the branch address. Where literal data is required as part of an instruction it follows in the succeeding bytes (figure 3).

Bit 7	Bit 6	Interpretation
1	1	Unconditional branch (BRA)
1	0	Conditional branch (BEQ)
0	1	Ordinary instruction plus return from subroutine (RTS)
0	0	Ordinary instruction

Figure 1: Bits 7 and 6 of an instruction specifies its type.

Bit 7	Bit 6	Bits 5 - 0
1	x	High part of branch address
0	x	Opcode

Figure 2: Bits 5-0 of an instruction either specify the high part of the branch address or the opcode.

	Byte 1 (bits 5-0)	Byte 2	Byte 3	Byte 4	Byte 5
Branch	14 bit branch address		-	-	-
Load.L	Opcode	32-bit literal			
Load.W	Opcode	16-bit literal		-	-
Load.B	Opcode	8-bit literal	-	-	-

Figure 3: Multi-byte instructions specify literal data (big-endian format).

2.3.4 Overall CPU architecture

The CPU comprises a datapath and a control unit[10].

The datapath holds the registers and computation components associated with the data held in the parameter and return stacks. The datapath is a passive entity in the sense that it does not contain any control logic or state information of its own. Rather it includes a network of multiplexers and other switches that route data between registers and through computation components in various configurations. The behavior of the datapath at any moment is entirely governed by a set of external control signals feeding to it from the control unit.

The control unit is built around a sophisticated finite state machine (FSM) that is responsible for reading program instructions from system memory, decoding those instructions, and then setting the control signals to the datapath as appropriate for the execution of each. The control unit is also responsible for adjusting the program counter (PC) so that program instructions are read from memory in the appropriate order taking into account program jumps and branches, dealing with interrupts and other exceptions, and supporting data transfers between system memory and the data path.

2.3.5 Execution pipeline

The architecture of the CPU is built around a three stage execution pipeline. The pipeline stages are as follows:

1. "FETCH OPCODE". The next instruction is read from SRAM at the current address of the program counter. The control unit identifies the instruction type and extracts the opcode.

2. "DECODE AND EXECUTE" The current opcode is decoded via microcode and the appropriate control signals are sent to the datapath. The datapath configures according to the control signals and the result of the computation becomes available in combinatorial logic.

3. "SAVE" The parameter and return stack registers and memory (i.e. the synchronous logic) are updated with the result of the computation performed by DECODE AND EXECUTE in the previous stage.

The operation of the pipeline is illustrated with a worked example in figure 4. In this example, as at CPU clock cycle #0 the program counter is pointing to memory address zero. The execution pipeline proceeds thus:

Component / clock cycle	Cycle #0	Cycle #1	Cycle #2	Cycle #3
Program counter	0			
Instruction byte		38		
Opcode		38		
Microcode			1210	
TOS_n (combinatorial logic)			0	
TOS (synchronous logic register)				0

Figure 4: Illustration of the execution pipeline for the CPU instruction FALSE, which places zero on the parameter stack

1. "FETCH OPCODE". On the rising edge of clock cycle #1 SRAM system memory reads the data byte at the memory address pointed to by the program counter and makes it available to the control unit where it is known as the instruction byte. In this example the instruction byte has a value of 38 (corresponding to the instruction

"FALSE" which places zero on the top of the parameter stack). During the same clock cycle combinatorial logic within the control unit identifies (based on bit 7 of the instruction byte) that this is an ordinary instruction and extracts the opcode from the instruction byte. In this case the opcode also has the value 38.

2. "DECODE AND COMPUTE". On the rising edge of clock cycle #2 SRAM microcode memory within the control unit takes the opcode as a lookup address and returns the corresponding microcode value. During the same clock cycle the combinatorial logic in the datapath is configured according to the microcode value through its control signals and the value of the computation becomes available at the output of the multiplexer TOS_n (figure 5). In this case the microcode value is 1210 (corresponding to a particular configuration of control lines that will cause the datapath to push a zero onto the top of the parameter stack).

3. "SAVE". On the rising edge of clock cycle #3, the value presented by the multiplexer TOS_n (i.e. the result of the computation in the previous pipeline stage) is written into the synchronous logic register TOS (figure 5). At the same time the current value of TOS is written into NOS, and the current value of NOS is pushed into the SRAM block that holds the remainder of the parameter stack.

The CPU has a throughput of one instruction per clock cycle for most instructions since each pipeline stage executes in a single cycle, thus on every clock cycle another instruction is completing execution. The CPU has a latency of three cycles since it takes three pipeline stages to execute each instruction in full. As with any pipeline design there is a tradeoff between the number of pipeline stages and the maximum feasible CPU clock frequency. Longer pipelines have less logic to execute at each stage, thus requiring less time and permitting a higher clock frequency, but at the expense of higher latency and the introduction of issues such as conflicts between instructions at the beginning and end of the pipeline. The N.I.G.E. Machine's pipeline was designed to ensure deterministic execute at all times (i.e. no conflict states, failed branch predictions, etc.) at the same time as a maximizing clock frequency subject to that constraint. In particular the design mixes SRAM access (which tend to be very fast) with combinatorial logic functions (which tend to be slower) in stages 1 and 2 to balance the overall load throughout the pipeline.

2.3.6 The CPU datapath: parameter stack

Figure 5 illustrates the parameter stack datapath.

The top-of-stack (TOS) and next-on-stack (NOS) storage locations are 32 bit hardware registers while the remainder of the parameter stack is implemented with a dedicated 2KB SRAM block. This SRAM block is dual ported and the second port is mapped to the CPU address space. (This is useful for implementing FORTH instructions such as PICK.) The datapath is directed from the control unit via a 14 bit wide signal generated from control unit microcode that drive a set of multiplexers in the datapath and determine the data flow. The main multiplexers controlling the parameter stack are as follows:

- The multiplexer TOSn selects the value for the update of the TOS register from one of eight computation units: addition/subtraction, logic operations, multipurpose, comparison, multiply, unsigned multiply, divide and unsigned divide.

- The multiplexer NOSn selects the value for update of the NOS register from: TOS, NOS (i.e. itself, no-update), the item below NOS in the parameter stack RAM, or an arithmetic value from one of the computation units.

- The multiplexer PSPn is responsible for updating the parameter stack (PS) pointer, which is a 9 bit address signal spanning 512 * 32 bit cells in 2KB SRAM. The PS pointer can be incremented (the stack grows by one item), decremented (the stack

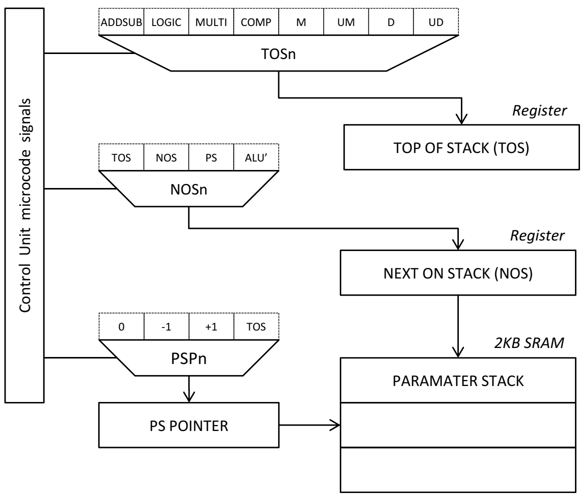


Figure 5: The parameter stack datapath illustrates the use of an SRAM block in combination with hardware registers.

shrinks by one item), held constant or updated with the current TOS value. When the PS pointer is incremented, the current NOS item is written from the 32 bit register to SRAM. The opposite dataflow occurs when the PS pointer is decremented.

The eight multiplexed computation units attached to TOSn essentially form the arithmetic logic unit (ALU) of the CPU. Each computation unit is directed by signals from the control unit microcode according to the functionality required by each operation. Some of the computation functionality is provided by Xilinx CORE modules that may leverage special purpose circuitry available within the FPGA such as hardware multipliers and carry logic structures. The computation units are summarized as follows:

- ADDSUB, an adder/subtractor implemented using a XILINX CORE template that leverages special purpose carry structures on the FPGA. There is a carry flag within the ADDSUB unit that is not directly accessible to the CPU but which gives the N.I.G.E. machine the capability to perform double precision addition and subtraction. The carry flag is only changed by one of the 7 addition or subtraction operations above and remains unchanged during the execution of all other instructions. (Interrupts should be temporarily suspended via the interrupt mask hardware register before performing double precision addition and subtraction to ensure that the hidden carry flag is not changed inadvertently).

- LOGIC, bitwise logic computations implemented in VHDL.

- MULTI, a general purpose multiplexer implemented in VHDL.

- COMP, a comparison unit implemented using a XILINX CORE template with supporting logic in VHDL.

- M and UM, signed and unsigned multiply implemented using on-chip FPGA pipelined multipliers with 32 bit operands and a 64 bit result. The operations complete in 5 clock cycles.

- D and UD, signed and unsigned divide implemented in logic fabric using a XILINX CORE template with 32 bit operands, a 32 bit quotient and a 32 bit remainder. The operations complete in approximately 40 clock cycles.

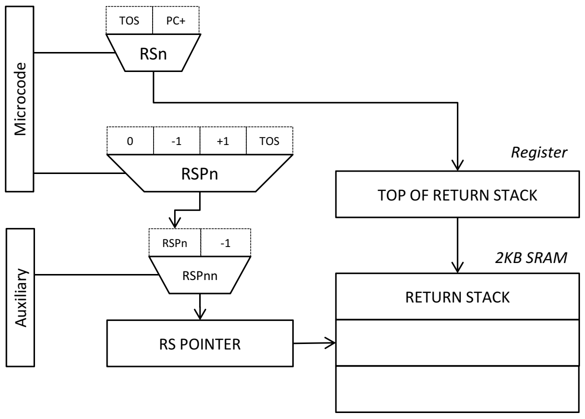


Figure 6: The return stack datapath illustrates the use of an SRAM block in combination with hardware registers.

2.3.7 The CPU datapath: return stack

Figure 6 illustrates the return stack datapath.

The Top of Return Stack (TORS) value is implemented as a 32 bit hardware register and the remainder of the return stack in a dedicated, dual ported 2KB SRAM block the second port of which is mapped to the CPU address space.

- The multiplexer RSn updates TORS with either the value from the top of the parameter stack (TOS), or the program counter of the next instruction following the instruction that is currently being executed. (The latter represents the operation of a JSR or BSR instruction.)
- The multiplexer RSPn updates the return stack pointer with either no change (0), decrement (-1, return stack size decreases), increment (+1, return stack size increases), or load from the parameter stack (TOS). The multiplexer is driven by a signal from control unit microcode.
- There is a secondary multiplexer, RSPnn driven by an auxiliary signal from the control unit that is able to decrement the return stack pointer regardless of the state of control unit microcode and the RSPn multiplexer. This is required because logic for the RTS instructions is hardwired rather than controlled by microcode.

2.3.8 The CPU Control Unit

The main components of the control unit are:

- A finite state machine (FSM) which determines next state logic and control signal outputs.
- Microcode held in a 2KB SRAM block that decodes instruction opcodes into control signals that will be routed directly to the datapath.
- A program counter and associated logic which steps program execution through memory in the appropriate order.
- Memory access logic which (a) routes memory write connections between the relevant bytes of the parameter stack registers and the appropriate system memory channels, and (b) accumulates byte or word length data from successive memory read cycles into a longword register which is connected to the datapath.

2.3.9 The finite state machine

The FSM is responsible for setting the values of control signals according to the current state and the current program instruction. Since the majority of CPU instructions execute in a single cycle, in most cases there is no change of state from instruction to instruction. The state in which all of the single-cycle instructions are executed is documented in the VHDL source code with the name "common". The state machine changes state from common to one of a number of other states for the following events:

- Instructions that take more than a single cycle to execute (?dup, multiply, divide, load literal, memory fetch, and memory store).
- Jumps, branches, and returns.
- Interrupts and traps.

2.3.10 Microcode

The CPU datapath requires 14 control lines to direct the various multiplexers and computation units appropriately for each instruction (plus one auxiliary control line for the RTS instruction). A simple "hardwired" decoder in the CPU control unit might require that these control lines be represented directly in the bits of the CPU instruction set. However by using microcode, the 14 control lines can be obtained from only 6 bits in the CPU instruction by configuring a 2K SRAM block with 6 address lines and 14 data lines. This enables higher code density through single-byte instruction encoding . There is a latency of one clock cycle for reading the microcode from the SRAM. This corresponds to part of the second stage of the pipeline ("DECODE AND EXECUTE").

2.3.11 Program counter

The control unit operates such that the code of the next instruction is being read from SRAM at the same time as the current instruction is being executed. This is part of the first stage of the pipeline ("FETCH OPCODE"). Update of the program counter is controlled by the FSM. At each cycle the possibilities for update of the program counter and return stack are:

- For single cycle instructions and load literal instructions, add one to the PC. Load literal instructions proceed byte by byte through the literal data using the PC.
- For other multi cycle instructions, add zero to the PC until the last cycle of the instruction and then add one. This is required to prime the first stage of the pipeline ("FETCH OPCODE") so that the next-but-one instruction is read from memory at the appropriate time
- For an external interrupt, redirect the program counter according to the vector number provided by the interrupt controller. In this case the current value of the program counter needs to be placed on the return stack since the current instruction will not be executed.
- For a TRAP or RTI_TRAP instruction, redirect the program counter to the trap vector. (The RTI_TRAP instruction is a two-phase instruction used for single stepping; first of all an RTI from the current trap routine is made, then one instruction at the current PC is executed, and then control is passed immediately back to the TRAP vector). For a TRAP instruction the PC of instruction following the current instruction is stored on the return stack.

- For a jump (JSR, JMP), redirect the program counter to the value currently on the top of the parameter stack (TOS). In the case of a JSR, also save the address of the next instruction on the return stack.

- For a branch instruction (BSR, BRA, BEQ), if the branch is taken redirect the program counter to the value of the PC plus the value on the top of stack. BRA and BEQ are two byte instructions and the PC will be on the second byte when the branch calculation is made. This needs to be taken into account by the assembler when calculating branch offsets.

2.3.12 Memory Channels

The CPU has three separate memory channels. Each of these channels has a read data bus, a write data bus plus memory control signals as required. A single address bus is common to all channels. The three data channels are as follows:

- An 8-bit data channel to SRAM.

- An 8-bit data channel to PSDRAM via the direct memory access (DMA) controller.

- An 16-bit data channel to PSDRAM via the DMA controller. This memory channel has twice the bandwidth of the 8-bit channel and is used for the read /write of word and longword data.

2.3.13 Other hardware components

The other principal hardware components implemented in VHDL are as follow:

Interrupt controller Responsible for prioritizing and scheduling interrupt signals from I/O devices to the CPU. The interrupt controller can be configured to accept additional interrupts from user-designed components in the digital logic layer that have their own interrupt vector routines. An interrupt mask register is available for enabling and suspending interrupts.

Video controller Responsible for providing a VGA signal for connection to a monitor. The video controller provides 256 colours and text/character graphics resolutions of 100*75, 100*60, 80*60, or 80*48 characters per screen, plus pixel graphics resolutions of 800*600 or 640*480 pixel per screen, double buffered.

Direct Memory Access (DMA) controller Responsible for multiplexing access from the CPU, the video controller, and other components to the 16MB PSDRAM on board the Nexys 2 development board.

RS232 controllers There are two by default, which provide an RS232 port for general purpose I/O and a dedicated RS232 port for connection to a third party SD-card reader/writer that serves as an external storage medium for the N.I.G.E. Machine.

PS/2 keyboard controller For direct connection to a PS/2 keyboard.

Memory-mapped hardware registers The interface between system control registers to the CPU address space. The hardware registers are expandable to accommodate user-designed components in the digital logic layer.

2.4 Implementation of system software

The N.I.G.E. Machine's FORTH environment is coded in assembly language and occupies just less than 8K of system memory. Published versions of FORTH were examined for guidance with the implementation[12][13]. For the most primitive FORTH words, there is a one-to-one correspondence with the CPU instruction set. Other FORTH words are implemented as machine language subroutines. There is no inner interpreter. The operating model on the N.I.G.E. Machine's FORTH environment could be classified as subroutine threaded or native.

Because the CPU instruction set is in general a subset of primitive FORTH words, the FORTH environment serves as the "local assembler" for the N.I.G.E. Machine. The N.I.G.E. Machine's FORTH implementation was developed in assembly language on a PC with a specially developed two-pass cross assembler and the cross assembler itself is written in standard ANSI FORTH.

The ANSI FORTH CORE[14] wordset has been implemented with very few exceptions, along with a selection of the most applicable words from the CORE EXTENSION, FACILITY, FILE ACCESS, PROGRAMMING TOOLS and STRING wordsets . Where minor departures from the ANSI standard have occurred they are due to reasons of implementation efficiency on an embedded system. In addition, a set of system specific words have been developed to enable convenient control of the N.I.G.E. Machine's facilities.

3 Results

3.1 Synthesis results obtained from the Xilinx ISE development software.

Version 13.2 of the Xilinx ISE tools was used to develop and synthesize the logic design for the Xilinx XC3S1200E Spartan-3E FPGA that is used in the N.I.G.E Machine. To put this FPGA in context, broadly speaking Xilinx has for several years offered two main families of device, Virtex and Spartan. Virtex are the high performance devices and Spartan are the economy or high-volume devices. Device families are also differentiated by generation numbers that indicate improving technology, typically driven by advances in the manufacturing process. Currently the latest devices in the Virtex family are at generation 7 and the latest devices in the Spartan family are at generation 6. Within the Spartan family, the prior generation to 6 was 3 (generation numbers 4 and 5 were skipped). Table 1 offers a comparison of Xilinx FPGAs according to the performance of the proprietary Xilinx softcore CPU, the MicroBlaze.

The Spartan-3E FPGA used in the N.I.G.E. Machine is therefore a one-generation-old device in the economy family of Xilinx FPGAs, and so relatively modest in comparison to the latest available technology. Nevertheless, it is in itself a highly capable device.

Device family	Typical MicroBlaze clock speed (3 stage pipeline format)
Virtex-6/7	240 MHz
Spartan-6	150 MHz
Spartan-3	50 MHz

Table 1: Xilinx device families compared according to MicroBlaze performance[15].

Resource	Used	Available	Utilization
4-input LUT's	3,884	17,344	22%
Slice flip flops	2,920	17,344	16%
2K block RAM	28	28	100%
Multipliers	8	28	28%

Table 2: N.I.G.E. Machine FPGA utilization on a Xilinx XC3S1200E, Spartan-3E FPGA.

Resource	LUT's
CPU	2,529
of which datapath	1,920
of which control unit	609
DMA controller	364
Hardware registers	280
Video controller	114
Diligent IO port	95
RS232 controller	71
Reset controller	52
PS/2 controller	39
System RAM	33
Interrupt controller	31

Table 3: N.I.G.E. Machine FPGA utilization at module level.

Parameter	
Maximum frequency	50.140 MHz
Minimum period	19.944 ns
Minimum input required time before clock	11.507 ns
Minimum output required time after clock	13.097 ns

Table 4: N.I.G.E. Machine timing summary on a Xilinx XC3S1200E, Spartan-3E FPGA.

3.2 Instruction frequency

Instruction	Frequency
LOAD.W	17.88%
JSR	9.17%
RTS and ,RTS*	9.06%
LOAD.B	6.47%
BEQ	4.60%
DUP	4.17%
OVER	3.67%
FETCH.L	3.56%
DROP	3.42%
STORE.L	3.02%
SWAP	2.91%
R>	2.37%
BRA	2.37%
FALSE	2.23%
FETCH.B	2.19%
1+	2.05%

* Of which RTS 6.36% and ,RTS 2.70%

Table 5: The 80% most used CPU instructions in the FORTH system software (as counted by the cross-assembler and ignoring execution frequency differences due to loops and conditional code, etc.)

3.3 Implications for design objectives

The FPGA logic utilization of 22% for a full micro-computer system on a relatively modest device is a very positive result. There is a significant amount of room remaining on the FPGA for the digital logic layer. If anything, further improvements in the design could be focused on addressing some of the current design limitations even at the expense of consuming a reasonable amount of additional logic area.

The maximum frequency of 50MHz was roughly as expected on this particular FPGA given the benchmark to the MicroBlaze (Table 1). Detailed analysis of the post place-and-route timing report did not reveal obvious bottlenecks in any one area of the design. The delays are roughly balanced between logic and routing. There was some evidence that the carry structure in the datapath adder unit may be a slightly slower path, and likewise the hardware registers.

The instruction frequency results were illuminating when considered in relation to the design objectives. The table well illustrates the load-store architecture of the CPU (LOAD.W is the most used instruction), and the subroutine threaded nature of FORTH (JSR and RTS are the second and third most used instructions). Given that high instruction throughput was specified as a design objective, it is interesting that the most used instruction (LOAD.W) is one of the minority of instructions that do not execute in a single cycle. (LOAD.W executes in three cycles as a direct result of the narrow instruction fetch that was discussed under design limitations.) Another optimization specified at the design stage, the compound RTS instruction, is only used in 30% of all return-from-subroutine instances, perhaps because of the limitation that it is only compatible with single cycle instructions that do not otherwise involve the return stack.

Clear priorities for future versions of the CPU softcore will be to widen the instruction fetch and to broaden the applicability of the compound RTS instruction, as well as identification of the hardware modules that can be further developed to increase the maximum potential clock frequency.

4 Application discussion

An experimental setup in applied physics served as an illustration of the use of the N.I.G.E. Machine. A short video is available to demonstrate [16]. The objective of the experiment was to measure the response of a certain light sensor to changes in the brightness of an LED. The light sensor in question was a light-to-frequency converter manufactured by the firm TAOS which comprises a photodiode and a current-to-frequency converter in a single package. The package has three connecting pins: 5V supply, ground, and output. The output signal is a square wave that varies in frequency from less than 1Hz to around 500kHz in response to the illumination of the photodiode. A tri-colour LED provided the illumination for the experiment. The LED has a common anode that connects to the positive supply voltage and three separate cathodes on the red, green, and blue elements that connect to ground via resistors of appropriate value. Three general purpose PNP transistors were connected between the cathodes and the resistors to provide switching for each colour element.

The circuit was constructed on breadboard. The output pin of the TAOS sensor and bases of the three PNP transistors were connected via hookup wires to an expansion port on the Nexys 2 board that routes directly to free pins on the FPGA.

A digital logic layer was designed to control and take measurements from the circuit in real time. As with the N.I.G.E. Machine overall, the Xilinx webpack tools were use for this development work. (The webpack tools are a free download from the Xilinx website[15].)

A frequency counter was required for measuring the output of the TAOS unit. A straightforward frequency counter module was developed in VHDL that comprised (a) a debounce process to eliminate any switching noise on the signal line, (b) a finite state machine to follow the square wave of the TAOS signal and (c) a counter and register to record the number of cycles of the square wave in one second. For controlling the three LED elements three variable duty cycle square wave outputs were required (with a variable duty cycle square wave, the average current to an LED element can be adjusted whilst keeping the supply voltage constant. If the square wave has a sufficiently high frequency (say 100Hz or more) there will not be any flicker observed by the light sensor.) A variable duty cycle square wave generator is very straightforward to design in VHDL and a suitable module was written in less than about 10 lines of logic description. Both the frequency counter and the variable duty cycle module were interfaced to the micro-computer layer of the N.I.G.E. Machine via hardware registers memory mapped to the address space of the CPU. This was accomplished by extending the existing hardware register module of the N.I.G.E. Machine. Three single-byte memory addresses were mapped to the duty cycles for the red, green and blue LED elements. Writing values of 0-100 to these registers adjusts the brightness of each of the red, green, and blue LED elements from full off to full on in real time. The output from the frequency counter was mapped to a longword memory address that could be read directly as the current frequency reading in Hz. The frequency reading in the register is automatically updated each one second in this design.

The digital logic layer was initially developed independently of the N.I.G.E. Machine in a stand-alone application. After a brief logic design was drawn up, the VHDL simulator was used to verify that the modules were operating as expected. A few small enhancements and simplifications were made at the simulation stage. After simulation was complete the Nexys2 board was programmed with the design of the two modules and connected to the breadboard and the electronic circuit. The modules were verified working as expected. At this stage a branch was made in the Subversion version control repository where all of the source code for the N.I.G.E. Machine is held. Using a branch structure in the version control system allowed a special version of the N.I.G.E. Machine source code to be created without affecting the main development path. The frequency counter and variable duty cycle modules were incorporated into the source code of the N.I.G.E. Machine and the Nexys2 board was programmed with the revised version. The functionality was again verified.

Finally, the FORTH environment was used to begin investigating the properties of the circuit. Initially small FORTH definitions were constructed to set the duty cycles of the three LED colour elements and read the frequency count from the TAOS sensor. These words were used to informally investigate such things as the dynamic range of the sensor/LED combination, the level of illumination background in a lit and unlit room, and the illumination levels of the three different LED colours. This general "tinkering" allowed the experimenter to gain a general feeling for the apparatus before running an actual experiment. Next, simple FORTH words were constructed to test the frequency output at various levels of duty cycle input and repeat these measurements over a series of input values. The results were read from the N.I.G.E. Machine and analyzed on a PC using Microsoft Excel. (The N.I.G.E. Machine includes an interface to a third party SD-card reader/writer than could also be used for logging the experimental results and transferring them to a PC for analysis.)

There are various ways in which the sophistication of the experimental set-up could be extended. For example the frequency counter module in the digital logic layer currently counts the signal from the light sensor over a period of one second. This is a simple and direct way to obtain a frequency count but a better dynamic range and/or response speed for measurements could be achieved by making the count period user selectable, for example 1/16 second, 1/4 second, 1 second, 4 seconds. This could be easily achieved with another writable hardware register that directs the frequency counter module accordingly. Or, the measurement system could be changed so that rather than counting the number of square waves over a fixed time period, the time taken to receive a certain number of waves would be measured instead, and thus the dynamic-range would be self-adjusting. An interrupt could also be used to signal when each new reading is ready.

The example described here illustrates the general approach of using a custom built digital logic interface for a new application and developing with a rapid prototyping environment. The initial digital logic interface functionality may be kept quite straightforward so as to minimize the time needed prepare the first design. As the user finds that there are experimental boundaries that need to be pushed back, it is straightforward to go back and iterate the design of the digital logic layer in a focused way to meet those goals. FORTH is used as a "language for direct communication between human beings and machines". The interactivity allows the experimenter to tinker with the apparatus at the initial stage, perhaps checking the overall characteristics or problem-solving any issues. When ready, the experiment proper can be built bottom up using the small routines that are already tested and understood.

5 Next steps and acknowledgments

The experiment described here was chosen just as an example to illustrate the capabilities of the N.I.G.E. Machine and the typical steps in developing an application. It is hoped that the N.I.G.E. Machine will find use in real scientific projects going forward. Future developments are anticipated that will enhance its capabilities including porting the design to more advanced FPGA development boards that offer additional speed and functionality (for example the Digilent Atlas board using a Spartan-6 FPGA).

I would like to acknowledge and thank Gunnar von Boehn and Jens Künzer of the Natami project for their thoughts on VHDL development and the N.I.G.E. Machine system manual, Jonathan Keelan of the Open University for his advice on the presentation of this paper, and the anonymous reviewers who's suggestions considerably improved its clarity of communication. The remaining deficiencies of the paper are solely the responsibility of the author.

References

[1] Volnei A. Pedroni, “Circuit Design and Simulation with VHDL”, 2nd edition, MIT Press, 2011

[2] The author, http://www.youtube.com/watch?v=0v-HuVLRoUc

[3] James Bowman , “J1: a small Forth CPU Core for FPGAs” in *EuroForth*, 2010

[4] K. Schleisiek, “MicroCore,” in *EuroForth*, 2001.

[5] B. Paysan, “b16-small – Less is More,” in *EuroForth*, 2004.

[6] E. Hjrtland and L. Chen, “EP32 - a 32-bit Forth Microprocessor,” in Canadian Conference on Electrical and Computer Engineering, pp. 518–521, 2007.

[7] E. Jennings, “The Novix NC4000 Project,” Computer Language, vol. 2, no. 10, pp. 37–46, 1985.

[8] Rible, John, "QS2: RISCing it all," Proceedings of the 1991 FORML Conference, Forth Interest Group, Oakland, CA (1991), pp. 156-159.

[9] Stephen Pelc, “Programming FORTH”, MPE, 2011

[10] Enoch O. Hwang, “Digital Logic and Microprocessor Design with VHDL”, C.L. Engineering, 2005

[11] P. J. Koopman, Jr., “Stack computers: the new wave”, Halsted Press, 1989

[12] Brad Rodriguez, “Moving Forth”, The Computer Journal, 1993

[13] Phil Burk, “pFORTH”, public domain, 1994-2012

[14] Elizabeth D. Rather and Edward K. Conklin, “Forth Programmer’s Handbook”, 3rd edition, Booksurge Publishing, 2008

[15] Xilinx website, http://www.xilinx.com

[16] The author, http://www.youtube.com/watch?v=0Kj5EMdnkMk

Forth for Education - 4E4th and 4E4th IDE

Dirk Bruehl
BRUEHLCONSULT
www.BruehlConsult.com

Abstract

This paper describes the necessity to offer special easygoing tools for the next generations of enthusiastic and engaged Forth programmers education, frankly reporting ways and means, showing a trailblazing example on the path to get there.

Preface

This is rather more an experience report than a technological review, but it has to be that way, to later make the conclusions better understandable for the reader.

Recently I received a TED Talk by Kevin Kelly [1]: How technology evolves.

I heard him saying "*today, there are millions of young children being born whose technology of self-expression has not yet been invented. We have a moral obligation to invent technology so that every person on the globe has the potential to realize their true difference*."

Forth has the ability to do it's part of this moral obligation for millions of young children who are already born - the Forth technology of self-expression has been invented and has to be fed into practice for this purpose. That's why I am proposing an Education Initiative giving the opportunity to teach and learn Forth in a new, up to now unknown, and easygoing fashion.

We call it 4E4th - Forth 4 Education.

How it began

The starting point – not long ago - was Texas Instruments offering the MSP-EXP430FR5739 Experimenter Board [2], having TI's first commercially available FRAM [3] microcontroller [4] on board. When in 1990 I first read about FRAM [5], I immediately recognized that this is the future of microprocessor memory - like in the seventies when I recognized CMOS, microprocessors, LEDs and LCDs to be the future of electronics.

Now there it was, the Forth Dream Machine, a microcontroller where you didn't have to worry about special programming and saving procedures, the first time that features which make PC programming easy could be transferred to microcontroller programming, especially useful in Forth. Two decades I had been waiting for this to happen.

I am a Forth user, not a Forth system writer. So I asked my friend Michael Kalus to help get a Forth running on this MSP-EXP430FR5739 Experimenter Board, and Michael ported Brad Rodriguez's MSP430 CamelForth [6] - removed the FLASH part and changed everything what was needed to make a pleasant and easy to use Forth out of it, a Forth where you could do incremental programming, no worries needed in case of power outage. When power is back, all data are still there because of the nonvolatile FRAM storing mechanism. Works great!

At that time I tried to transfer an audio project which I had done with TI's MSP430F2012 [7] - for price and power reasons, a consumer gadget - to TI's MSP430G2553 [8], the new "Value Line" [9] microcontroller, which should allow my customer to program some audio features by himself.

The MSP430F2012 I had to program using C - which I didn't like. I had to do so because of memory constraints. Not enough memory for Forth. Now I tried to get my program running on the new MSP430G2553, but to no avail. TI had changed a lot of I/O functions, and even that I only needed basic functions like ADC, SPI, and PWM, I couldn't get it running. And the user programmability demanded to have a Forth kernel. I asked Michael again, this time to port Brad Rodriguez's MSP430 CamelForth to the MSP430G2553.

When done, we recognized that this was the right deal: TI's LaunchPad [10], a small inexpensive board, flashing Forth on it, very useful to teach basic microprocessor training in Forth - a dream Michael and others have had since a while: "Recently we had a meeting in the Lower Rhine region dreaming about a small nice affordable board with a modern MCU and a compact Forth inside" Michael later remembered.

That was just a few weeks before the 2012 annual Forth meeting of the German Forth Chapter, the "Forth-Gesellschaft e.V." Since $4.30 at that time have been 3.40€, we could offer a 4€-Forth selling the LaunchPad, despite the price of $1.20 we had to pay for the MSP430G2553, which was not part of the LaunchPad then.

That's how the idea started: a 4€-Forth.
When ordering our website it was immediately clear that a 4€4th website was not available because of character constraints. So www.4e4th.eu was born, and with it the idea to start an European Education Initiative; 4E4th standing for Forth 4 Education. The Forth meeting at Beukenhof, the Netherlands [11], was a first true European start and a great success. I managed to get fifty LaunchPads from three different sources to be delivered just in time and everybody at this meeting bought a 4€4th board - TI's LaunchPad with 4E4th inside.
To my surprise most of these LaunchPads have been delivered loaded with the MSP430G2553, which we needed. TI had changed the LaunchPad design meanwhile to fit our demands. The luck of the courageous, I would say.
4E4th was born.

4E4th for the LaunchPad - Forth for Education under the Hood

This version of 4E4th for the TI Launchpad with MSP430G2553 is based on CamelForth MSP430 V0.3 [12], created by Brad Rodriguez for TI 's MSP430F1611. He wrote: "This is an ALPHA TEST version of CamelForth/430, an ANSI Standard Forth for the Texas Instruments MSP430 family of microprocessors. This means that I have tested the bulk of this code for correct functioning, but you may still discover bugs. I'd appreciate hearing of any such by email at bj@camelforth.com.... CamelForth should be usable with any MSP430 device having at least 512 bytes of RAM, 8K of ROM, and one USART." [13]

This is the Memory map now:

```
0000-01FFh: Peripherals
0200-03FFh: 4E4th RAM, 512 bytes (stacks, buffers, variables)
1000-10FFh: "Information" Flash ROM (used for special variables)
C000-DFFFh: Program Flash ROM for new 4E4th definitions
E000-FFFFh: 4E4th kernel
```

The MSP430 uses a von-Neumann architecture — all program, data memory and peripherals share a common bus structure, consistent CPU instructions and addressing modes are used [14], too, ideally suited for Forth.

New 4E4th definitions are compiled into the Program (Instruction) space. New data structures (e.g., Variables) are allocated in the Data space [15]. 4E4th compiles source code directly into the MSP430's Flash memory.

The project has been created using TI's offer of IAR Kickstart [16].

4E4th is equipped with Autostart. The reset procedure executes BOOT. If there is a valid application it will start with this application. Otherwise WIPE is executed which cleans the FLASH to have a fresh start.

After executing SAVE the Forth state will survive a reset. Pressing and holding button S2 during reset will force a WIPE. This way 4E4th can be resurrected from a deadlock [17].

First Experience with starting 4E4th

The first steps to get 4E4th onto TI's LaunchPad have been using IAR's Embedded Workbench Kickstart, a task to meticulously follow Brad's eleven steps [15] from starting the Workbench to downloading 4E4th onto the target board.

Everybody who is following this way has to install IAR's Embedded Workbench Kickstart, a huge monster, occupying nearly one Gigabyte of disk space altogether. People who don't know 4E4th have to do this anyway [18].

Looking for less space and time consuming opportunities Michael discovered Elpotronics free FET-Pro430 Lite Software [19] and furnished the 4e4th.a43 file which reduced the LaunchPad flashing to less than a minute. To be able to do so, a special TI software from another TI source has to be started to prepare the USB to recognize the LaunchPad. Still a to do list.

We have been looking for ways to streamline the whole process, from flashing 4E4th to TI's MSP430G2553 LaunchPad, over getting interactively connected to 4E4th on the LaunchPad hardware; typing words, downloading 4th-files, and even starting and maintaining 4th-Projects. That's where our 4E4th Terminal-IDE comes in.

Four Steps towards the 4E4th Terminal-IDE

The 4E4th Terminal-IDE is the result of nearly thirty years of programming experience, using Forth in different projects and writing tools over this time to make life easier.

I have to share a short summary of circumstances which led to the idea for the 4E4th Terminal-IDE.

Step 1: Writing a virtual microprocessor engine

In 1979, using a KIM in Euro PCB format, I started a project with video output and database features [20].

In those times I wrote my programs with paper and pencil, reading each command from an Instruction Set Summary Card and using a hex key pad to get these commands into the microprocessors memory. To make complex programming easier, I always wrote little subroutines that could be tested easily and complete (years later I learned this is called structured programming [21]), so the video output and database routines have been a collection of subroutines. Typing each subroutine with six keystrokes soon was boring and I looked for better and faster ways to get this done.

Nearly two hundred subroutines had been accumulated, and easily I could write a table with all these starting addresses - reducing the number of keystrokes in half - because now I only had to type the number of the subroutine without adding any command. A little virtual engine took care of taking the subroutine and executing. That worked really great and totally fast. This was the start to write my own programming environment.

But then the August 1979 BYTE magazine featuring Forth [22] came in and I recognized: there it is, already done, and even better than mine, using words instead of numbers.

It took five years until I got my first Forth.

Step2: Starting with RSC-Forth

In 1984 I started with RSC-Forth [23], and it worked from start as expected.

Randy Dumse [24] had done a great job.

With RSC-Forth I was still able to work in a system I was used to, but in a much better way. RSC-Forth made it possible to add a 3” floppy while developing. I wrote my first own tool, a screen editor, not to depend on the command line input RSC-Forth provided with ending up typing lines again and again after typos.

There is only one thing with Forth which always bothered me:

There is a direct input mode, typing words and definitions directly to test these words, and then I had to type these definitions into the editor again to be saved and accumulated for the final download finishing the application. Doubling the workload again.

Having my first laptop some years later made the in system 3” floppy obsolete and made a safe download possible. Checking each byte echoed by RSC-Forth and stopping download when an error occurred, gave the possibility to jump automatically into the editor just at the line and character where the download error came up. This special communication program I had written in F-PC.

Step3: Fulfilling a dream: visualFORTH

When I worked for Lucent Technologies in 2000, I had to use Visual Basic [25] to write a programming tool for their production. To me VB was fascinating, different to plain text programming which I was used to. But it was not Forth, and therefore after a while it was boring.

Nearly ten years I waited for a visual Forth, not finding anything like this. There was a tool, associated with Win32Forth [26] - named ForthForm [27]. It was designed to make GUIs – which was a huge task, I am sure of that – but I didn't see how to get something working with this. I didn't recognize how to make a button to do some work when clicking on it.

So I started to upgrade ForthForm to make it usable for me and others [28], finally naming the result “visualFORTH” [29] - but that's another story.

VisualFORTH was the last step needed to make the 4E4th Terminal-IDE possible.

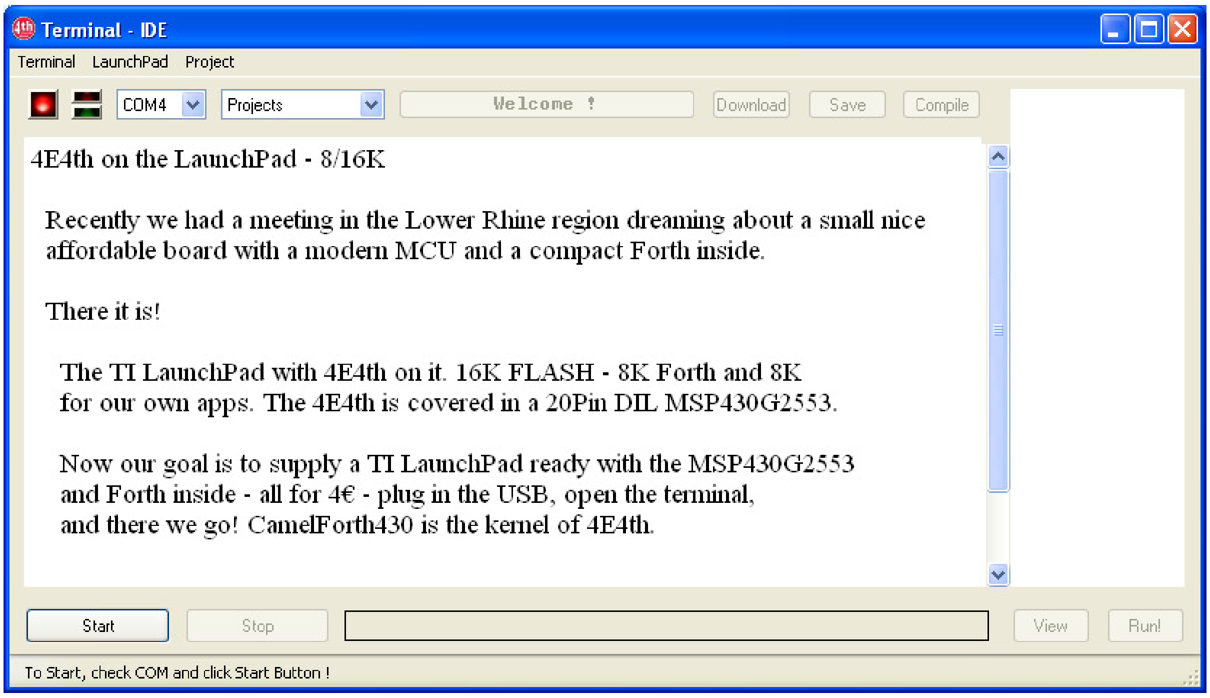
Step4: The 4E4th Terminal-IDE

At this point I will switch to a series of pictures, because “A picture is worth a thousand words” [30], they say.

There is one simple reason to do so: to show the simplicity of an IDE tool specially developed for Education in mind, reducing learning time to a minimum. The focus of the user should be on gaining experience in programming, avoiding all unnecessary distractions.

Starting the Terminal-IDE

The 4E4th Terminal-IDE needs no installation. Unpack and start right away!



4E4th on the LaunchPad - 8/16K

Recently we had a meeting in the Lower Rhine region dreaming about a small nice affordable board with a modern MCU and a compact Forth inside.

There it is!

The TI LaunchPad with 4E4th on it. 16K FLASH - 8K Forth and 8K for our own apps. The 4E4th is covered in a 20Pin DIL MSP430G2553.

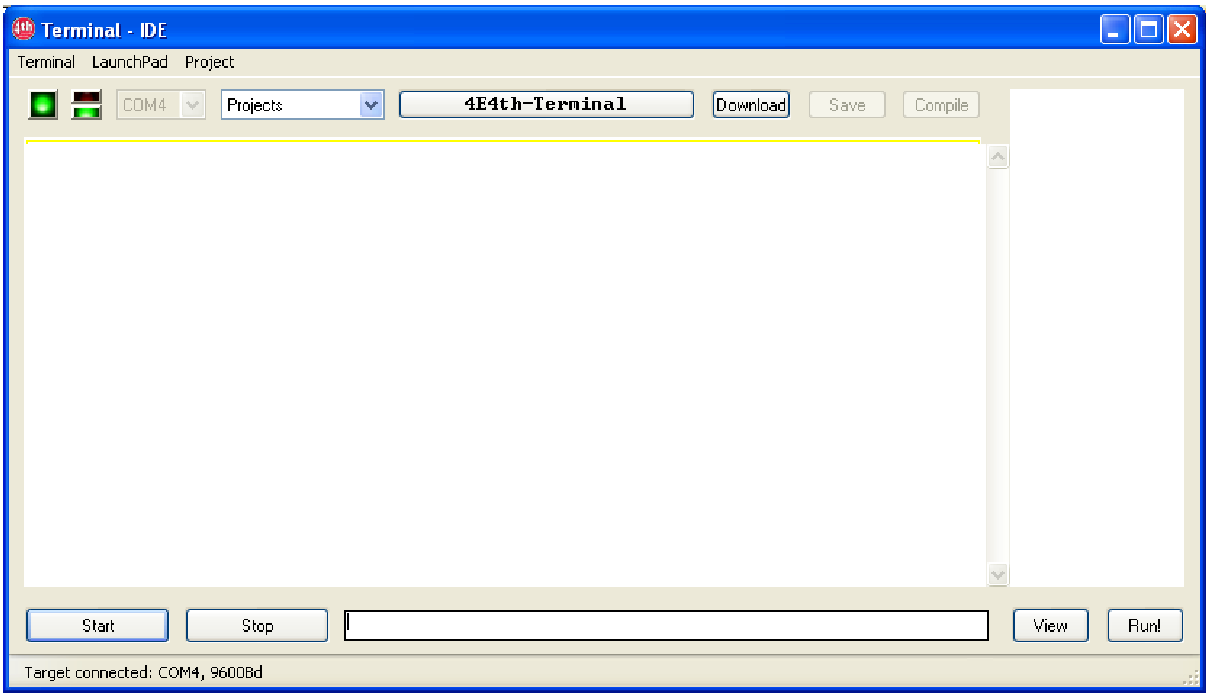
Now our goal is to supply a TI LaunchPad ready with the MSP430G2553 and Forth inside - all for 4€ - plug in the USB, open the terminal, and there we go! CamelForth430 is the kernel of 4E4th.

To Start, check COM and click Start Button !

The first appearance of the 4E4th Terminal-IDE is mainly a screen and a few buttons, following Dick Pountain's philosophy of “Simpleware” [31]: “Buttons … but no more than six of them.” And: “Things that are not active should go grey: live things go black again.”

At start a red LED and only one button to start with:

Now the target is connected:

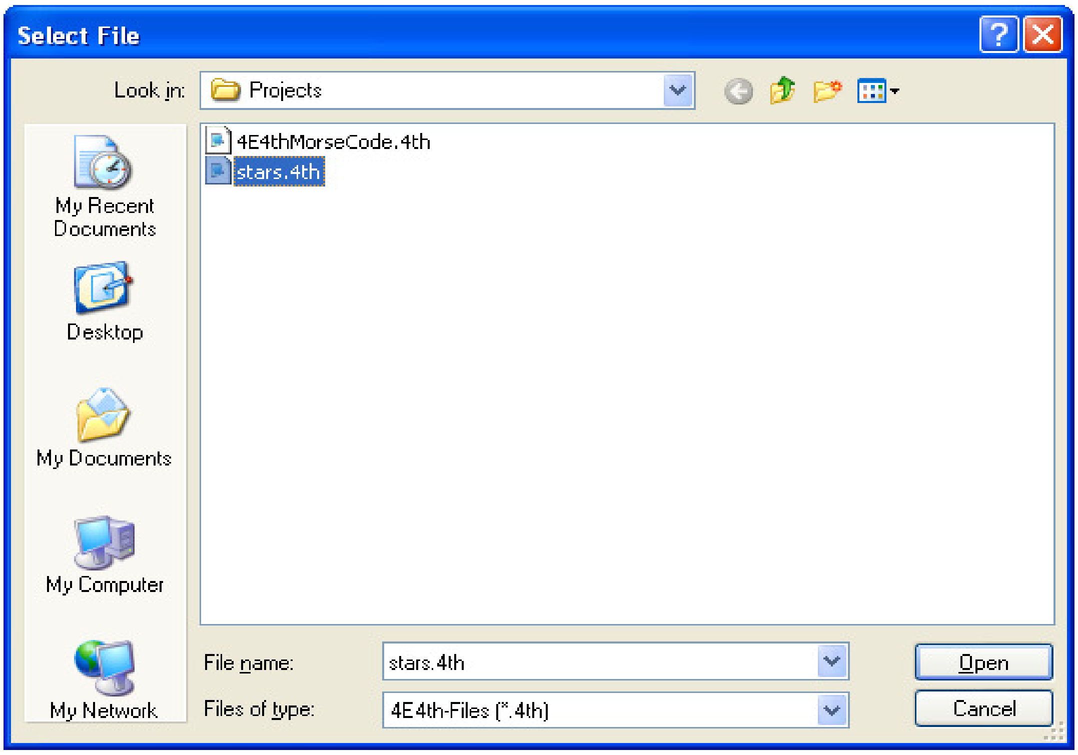


Target connected: COM4, 9600Bd

The LaunchPad USB Interface is always set to 9600Bd, 8N1. 4E4th Terminal-IDE checks this connection and connects to the COM port used by this USB.

Downloading a File

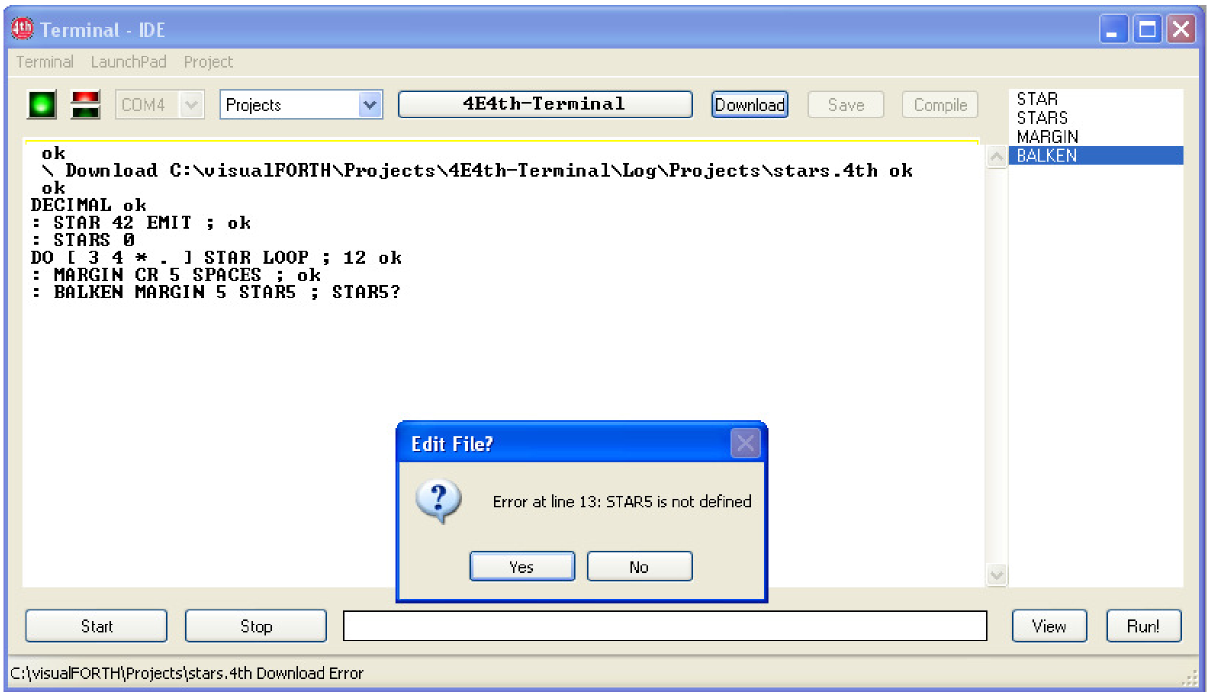
Click on “Download” opens a file-selection window, downloads with click on “Open”:



4E4thMorseCode.4th
stars.4th

File name: stars.4th
Files of type: 4E4th-Files (*.4th)

Download started. An error occurred while downloading:



```
 ok
 \ Download C:\visualFORTH\Projects\4E4th-Terminal\Log\Projects\stars.4th ok
 ok
DECIMAL ok
: STAR 42 EMIT ; ok
: STARS 0
DO [ 3 4 * . ] STAR LOOP ; 12 ok
: MARGIN CR 5 SPACES ; ok
: BALKEN MARGIN 5 STAR5 ; STAR5?
```

STAR
STARS
MARGIN
BALKEN

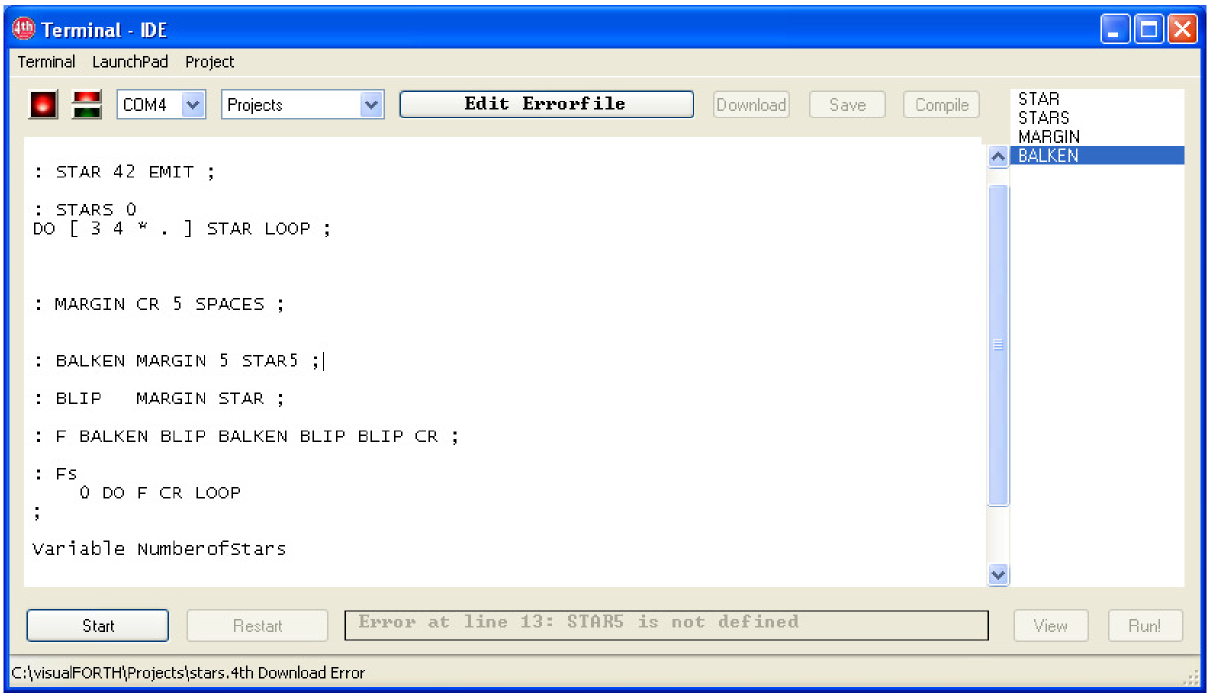
Edit File?

Error at line 13: STAR5 is not defined

Yes No

C:\visualFORTH\Projects\stars.4th Download Error

A click on “Yes” switches to “Edit Errorfile” mode, the cursor is at the error line:

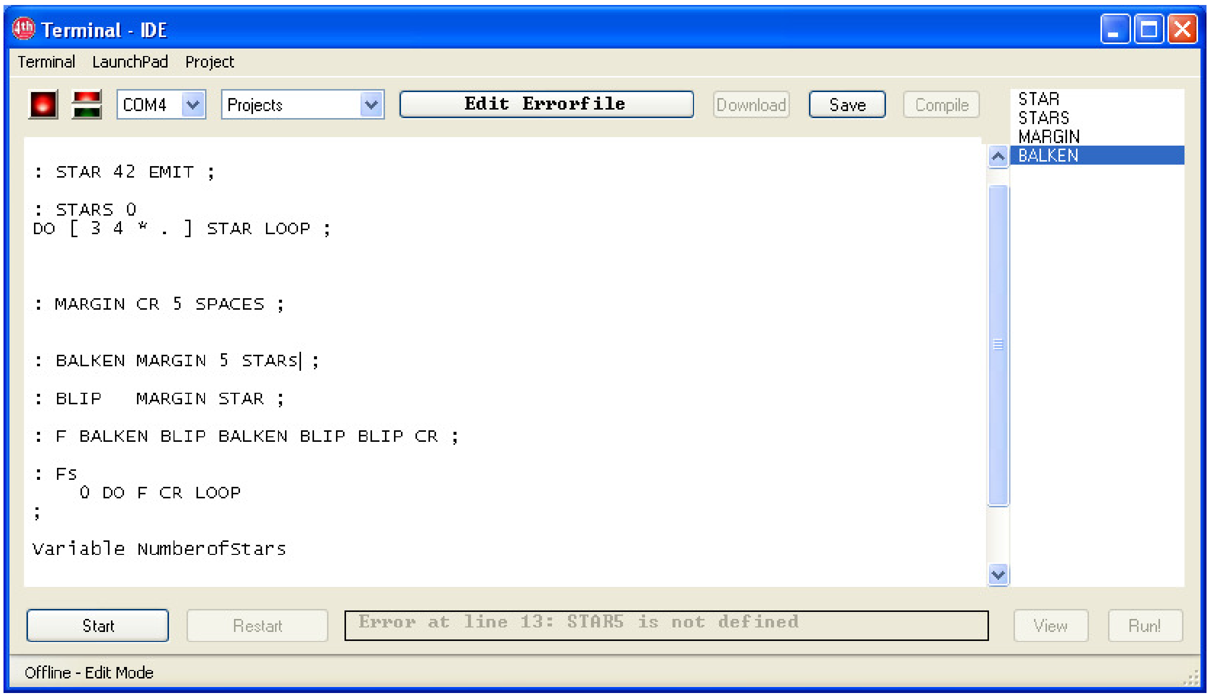


```
: STAR 42 EMIT ;

: STARS 0
DO [ 3 4 * . ] STAR LOOP ;


: MARGIN CR 5 SPACES ;


: BALKEN MARGIN 5 STAR5 ;|

: BLIP   MARGIN STAR ;

: F BALKEN BLIP BALKEN BLIP BLIP CR ;

: Fs
    0 DO F CR LOOP
;

Variable NumberofStars
```

```
Error at line 13: STAR5 is not defined
```

C:\visualFORTH\Projects\stars.4th Download Error

“Download”, “Save”, and “Compile” button are disabled. The 4E4th Terminal-IDE waits for error correction, so something has to be typed in to make the “Save” button work. The best thing of course is to correct this line:



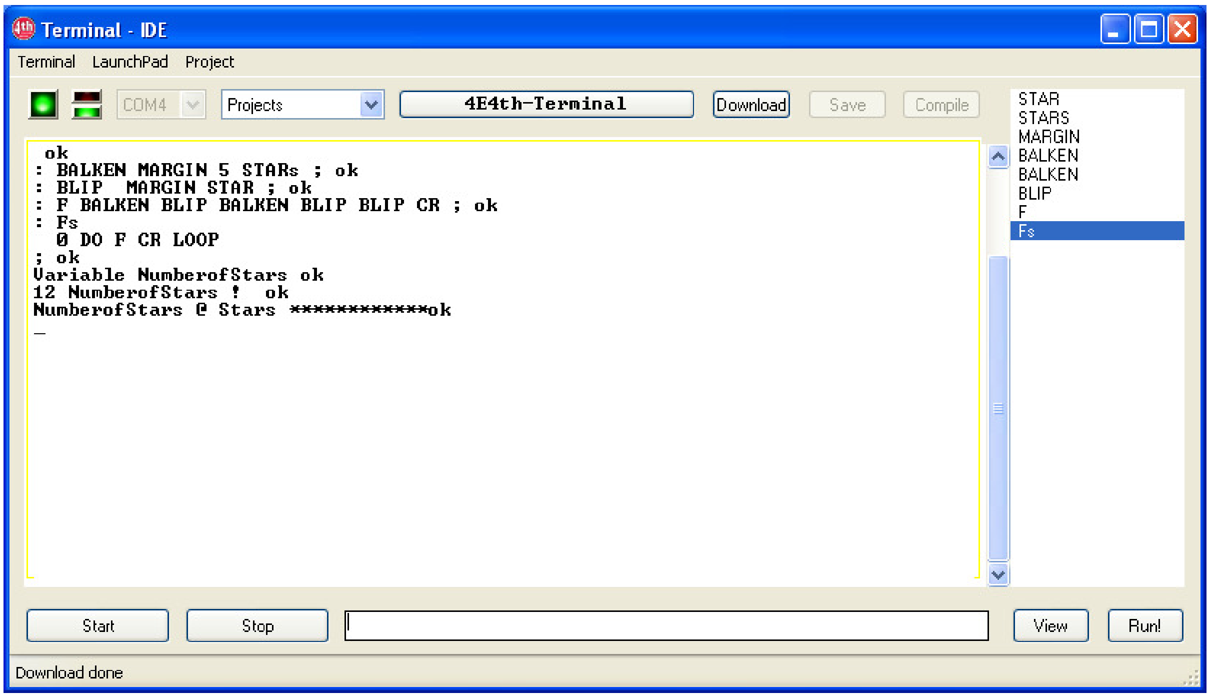
```
: STAR 42 EMIT ;

: STARS 0
DO [ 3 4 * . ] STAR LOOP ;


: MARGIN CR 5 SPACES ;


: BALKEN MARGIN 5 STARS| ;

: BLIP   MARGIN STAR ;

: F BALKEN BLIP BALKEN BLIP BLIP CR ;

: Fs
    0 DO F CR LOOP
;

Variable NumberofStars
```

```
Error at line 13: STAR5 is not defined
```

Offline - Edit Mode

When the correction is done, a click on “Save” saves this file, and a click on “Compile” downloads the needed parts of this file to the target to be compiled.

The 4E4th Terminal-IDE checks the last valid download line, and the download proceeds with the corrected definition, time saving real incremental compiling:

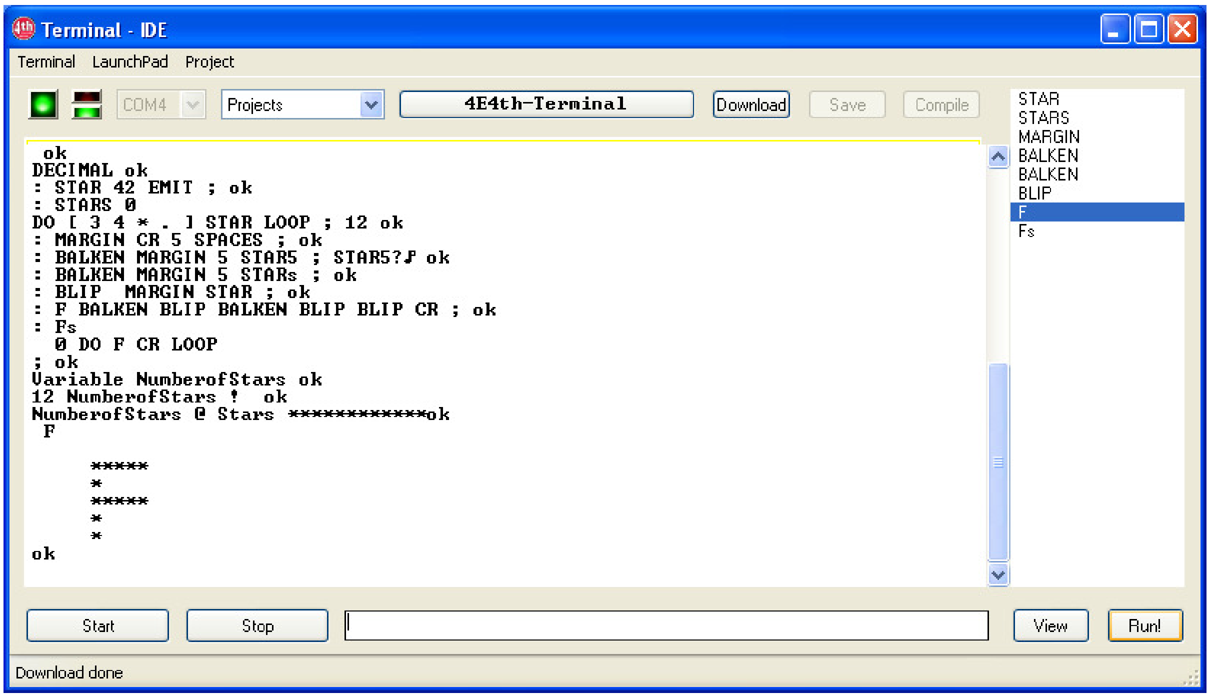


```
 ok
: BALKEN MARGIN 5 STARs ; ok
: BLIP  MARGIN STAR ; ok
: F BALKEN BLIP BALKEN BLIP BLIP CR ; ok
: Fs
  0 DO F CR LOOP
; ok
Variable NumberofStars ok
12 NumberofStars !  ok
NumberofStars @ Stars ************ok
_
```

Download done

Running a Program

Now we can test the program by highlighting the function to be tested and click on "Run!"



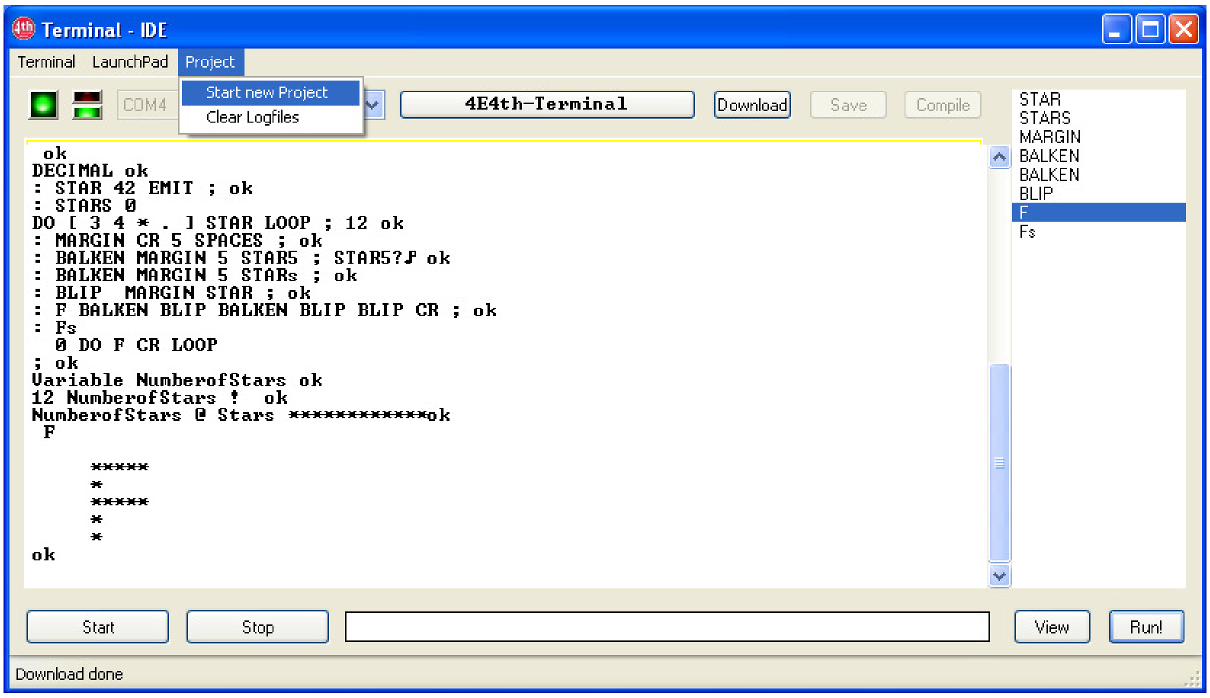
```
 ok
DECIMAL ok
: STAR 42 EMIT ; ok
: STARS 0
DO [ 3 4 * . ] STAR LOOP ; 12 ok
: MARGIN CR 5 SPACES ; ok
: BALKEN MARGIN 5 STAR5 ; STAR5?♪ ok
: BALKEN MARGIN 5 STARs ; ok
: BLIP  MARGIN STAR ; ok
: F BALKEN BLIP BALKEN BLIP BLIP CR ; ok
: Fs
  0 DO F CR LOOP
; ok
Variable NumberofStars ok
12 NumberofStars !  ok
NumberofStars @ Stars ************ok
 F

     *****
     *
     *****
     *
     *
ok
```

Download done

Voilà!

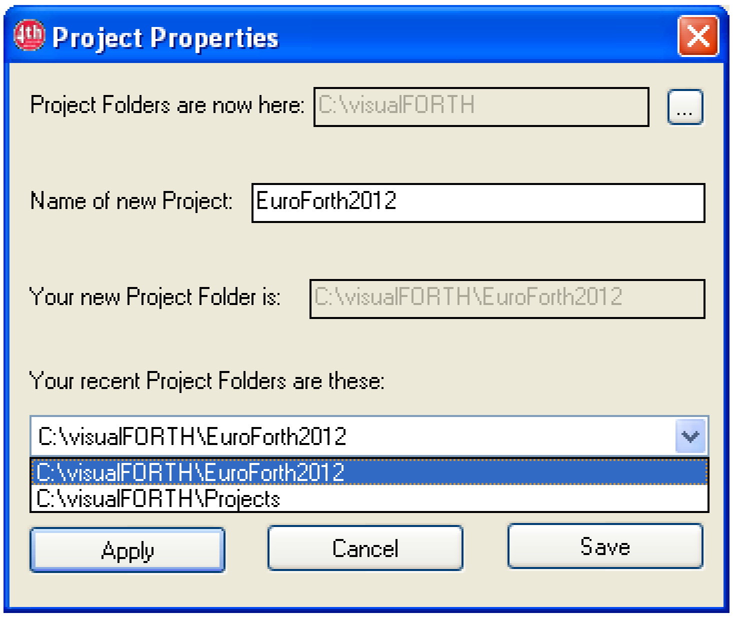
Starting a new Project

Like to start another project? Nothing simpler than that!
Dick Pountain wrote “Menus are poison”, but sometimes they are really useful:



```
 ok
DECIMAL ok
: STAR 42 EMIT ; ok
: STARS 0
DO [ 3 4 * . ] STAR LOOP ; 12 ok
: MARGIN CR 5 SPACES ; ok
: BALKEN MARGIN 5 STAR5 ; STAR5?♪ ok
: BALKEN MARGIN 5 STARs ; ok
: BLIP  MARGIN STAR ; ok
: F BALKEN BLIP BALKEN BLIP BLIP CR ; ok
: Fs
  0 DO F CR LOOP
; ok
Variable NumberofStars ok
12 NumberofStars !  ok
NumberofStars @ Stars ************ok
 F

     *****
     *
     *****
     *
     *
ok
```

To start a new project, simply use the “Project” menu, click on “Start new Project” and type the name of your new project into the blank line of the “Project Properties” pop up:



Project Folders are now here: C:\visualFORTH

Name of new Project: EuroForth2012

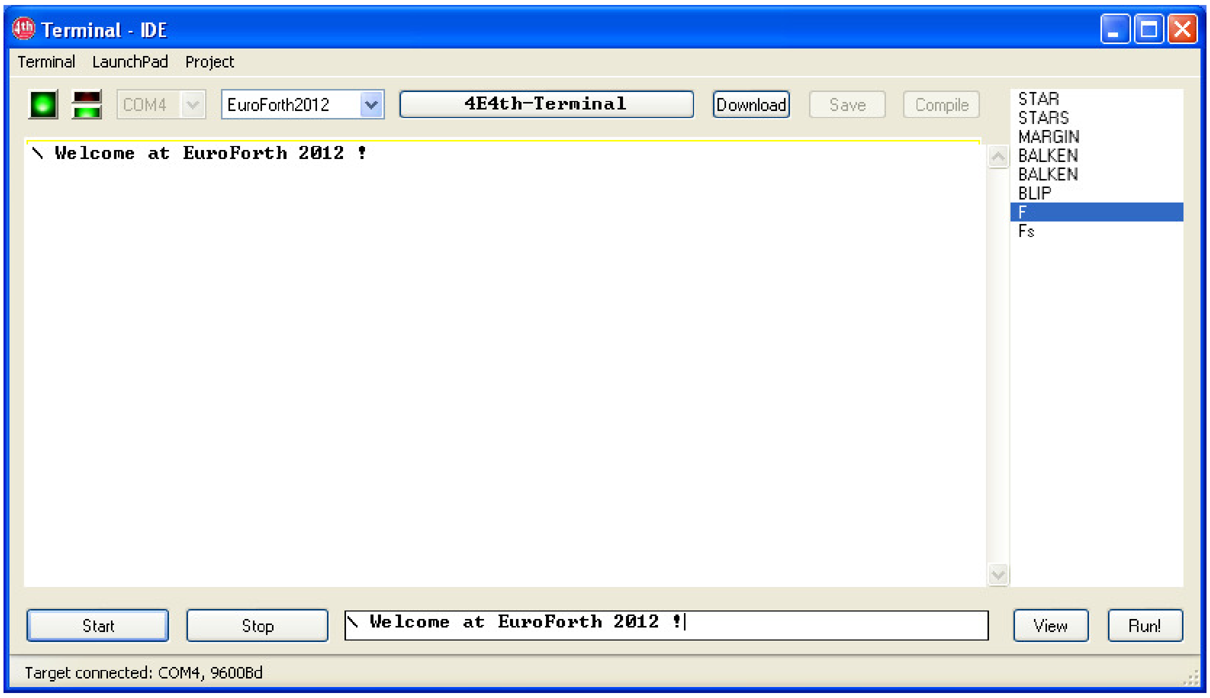
Your new Project Folder is: C:\visualFORTH\EuroForth2012

Your recent Project Folders are these:

C:\visualFORTH\EuroForth2012
C:\visualFORTH\EuroForth2012
C:\visualFORTH\Projects

Click on “Apply” and “Save”, and 4E4th Terminal-IDE starts your new project with a blank screen, ready to type in commands for the microprocessor.

Only a comment is shown here: “Welcome at EuroForth 2012”



```
\ Welcome at EuroForth 2012 !
```

```
\ Welcome at EuroForth 2012 !
```

Target connected: COM4, 9600Bd

The definitions are still there, as long as they are not wiped off.

To me the fascinating thing is that the user doesn't need to care about files.
The 4E4th Terminal-IDE takes care of all project files.
Nothing gets lost, and all previous works and changes are stored at a safe place.
A session may be interrupted and started at the same point again another day.

And imagine, the 4E4th Terminal-IDE occupies only 1MB disk space, including all necessary software to flash a new LaunchPad – the IAR-IDE, which is recommended to use with the MSP430, needs 1000 times as much!

This is only the beginning. Forth gives the ability to add other features over time. Learning with the user the toolmaker is able to sharpen his tools.

Conclusions

This experience report, including some technological review, has been done to make aware of some very important problems:

Knowledge accumulated over the centuries is diminishing from the conscience of people and accumulating nearly only on computer memories.

More and more programmers are coming directly from school or university, not having much experience of real life. They are used to producing wonderful pictures on computer screens but not caring about the experience of users who like to get a task done on a computer instead of waiting for animations to be downloaded.
That's my observation.

But the main thing is that these programmers have to be elite programmers because of the complexity of the products they are working on. My observation is that the number of elite programmers doesn't grow – the crew of elite programmers moves forward with the advances of technologies - leaving the others behind.

No one should be left behind. My goal is to give ordinary people the chance to do wonderful works of creation by themselves by offering a tool and a programming language which is reduced to a minimum of effort to use it.

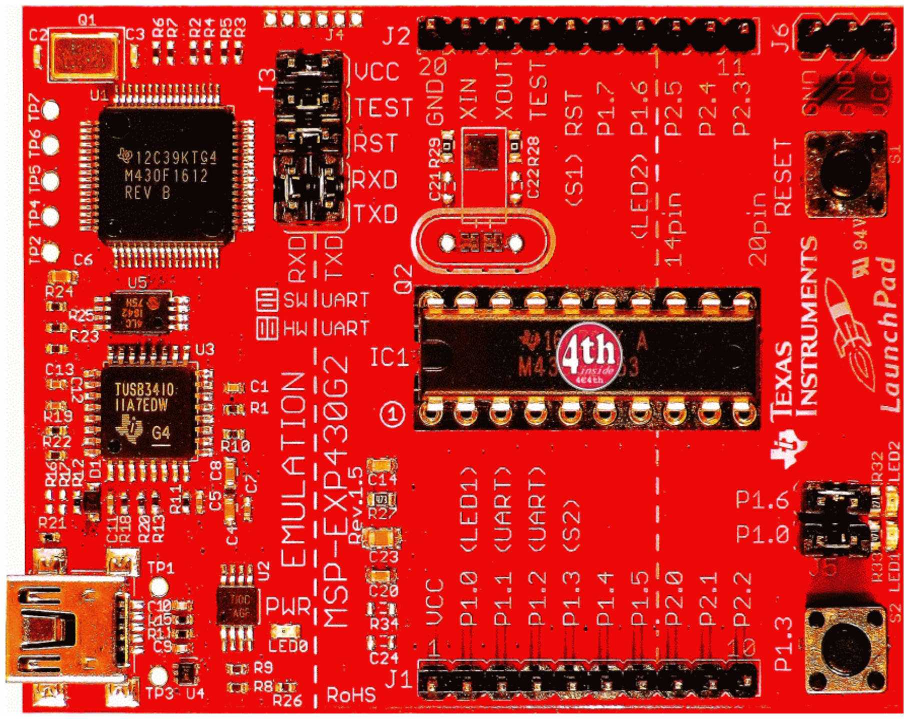
Forth is the programming language which stripped off all unnecessary attachment and distraction leaving pure logic to work with. Our 4E4th Terminal-IDE replicates that tradition, making real programming fun and easy.

I encourage everybody to think about that. Our approach is not patented, it is free to be copied. The intended goal to help young people to achieve self realization by doing great things should always be in mind. I don't have any idea what they will do with it - I trust our new generations to do the best for mankind. For mankind to survive, for example. Only Forth programs have the inherent ability to be readable and understandable even after centuries.

With 4€ they may start – here in the heart of Europe.

A bunch of experiments with 4E4th are offered at http://www.forth-ev.de/wiki/doku.php/projects:4e4th:4e4th:start:msp430g2553_experimente (waiting for translation).

It was never more affordable and easier to start a programming education – self education or school education. This situation should be taken advantage of by delivering the right tools to enable programming education, discarding all unnecessary obstacles, to make it a Newbees [32] delight.

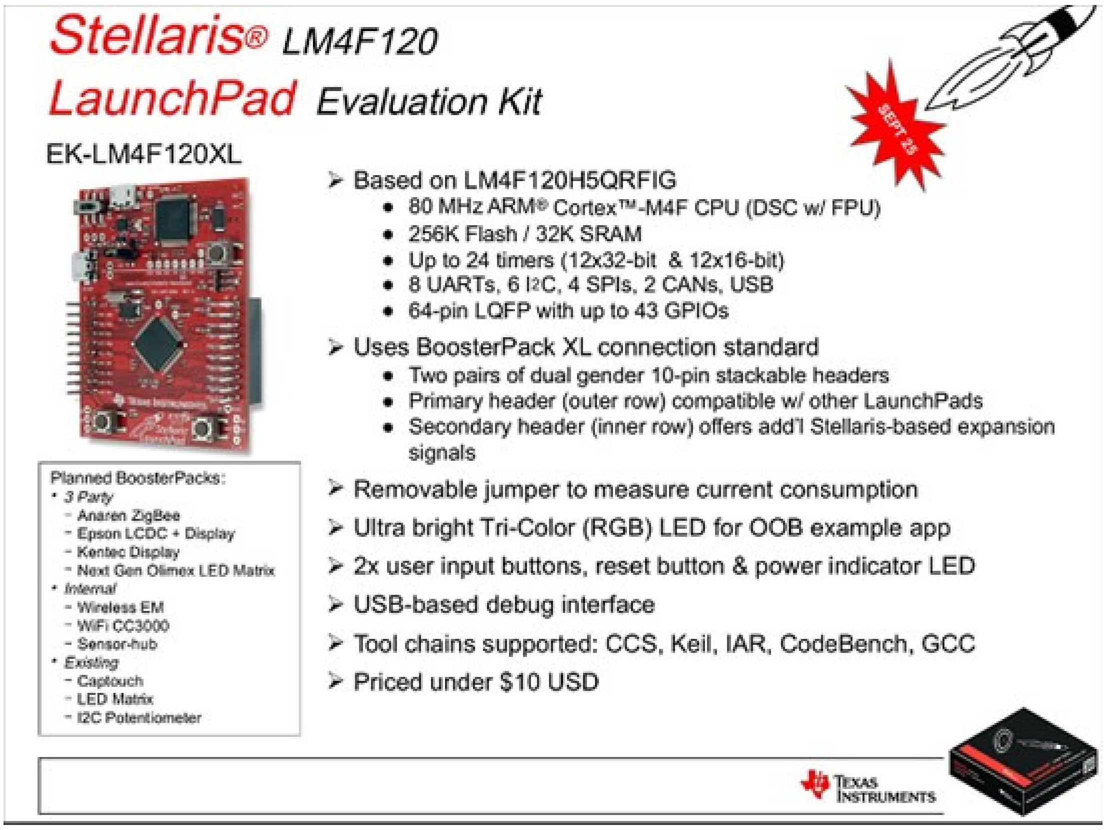


LaunchPad with 4E4th

A next Step

Here a small glimpse into the future:

TI's Stellaris LaunchPad at the promotional price of $4.99 [33] – another 4€-Forth promotional opportunity - is coming soon, announced for taking regular orders on September 25th – ten days from today:



Stellaris® LM4F120
LaunchPad Evaluation Kit
EK-LM4F120XL

SEPT 25

- Based on LM4F120H5QRFIG
  - 80 MHz ARM® Cortex™-M4F CPU (DSC w/ FPU)
  - 256K Flash / 32K SRAM
  - Up to 24 timers (12x32-bit & 12x16-bit)
  - 8 UARTs, 6 I2C, 4 SPIs, 2 CANs, USB
  - 64-pin LQFP with up to 43 GPIOs
- Uses BoosterPack XL connection standard
  - Two pairs of dual gender 10-pin stackable headers
  - Primary header (outer row) compatible w/ other LaunchPads
  - Secondary header (inner row) offers add'l Stellaris-based expansion signals
- Removable jumper to measure current consumption
- Ultra bright Tri-Color (RGB) LED for OOB example app
- 2x user input buttons, reset button & power indicator LED
- USB-based debug interface
- Tool chains supported: CCS, Keil, IAR, CodeBench, GCC
- Priced under $10 USD

Planned BoosterPacks:
- *3 Party*
  - Anaren ZigBee
  - Epson LCDC + Display
  - Kentec Display
  - Next Gen Olimex LED Matrix
- *Internal*
  - Wireless EM
  - WiFi CC3000
  - Sensor-hub
- *Existing*
  - Captouch
  - LED Matrix
  - I2C Potentiometer

Texas Instruments

Let's do the right steps now. Let's start a real **Forth Programming Education Initiative** to reach out to young people, even kids, giving them the opportunity to learn how easy they will be able to express themselves and unfold their abilities. To quote Andrew Reid [34]: “With your 4E4th we can show people how clever they can become.”

One last word:

Kevin Kelly [1] stated another remarkable truth: "Technologies don't die."

So relax, Forth will live forever.

Acknowledgements

My special thanks go to Michael Kalus, whose efforts brought 4E4th into life, and of course to Brad Rodriguez, who created CamelForth, upon which 4E4th was build.

A praise to Texas Instruments, especially to the people in Freising [35], Germany, who created these extraordinary power saving MSP430 microprocessors.

A great thank you of course to Chuck Moore, the genius behind all Forth inventions, and many thanks to Andrew Reid, who encouraged me to present this paper.

Many thanks to the Win32Forth crew, especially to Tom Zimmer, Ezra Boyce, Jos van de Ven, Camille Doiteau, Dirk Busch, Rod Oakford for preparing the ground to build 4E4th Terminal-IDE on.

And last, but not least, many thanks to Barbara Bruehl, who supported my endeavor, especially with proof reading.

References

[1] Kevin Kelly: How technology evolves, TED TALKS, 2006
http://www.ted.com/talks/kevin_kelly_on_how_technology_evolves.html

[2] MSP-EXP430FR5739 Experimenter Board, Texas Instruments Inc., 2012
http://www.ti.com/tool/msp-exp430fr5739

[3] Ferroelectric RAM, WIKIPEDIA, 2012
http://en.wikipedia.org/wiki/Ferroelectric_RAM

[4] F2MC-8FX Family FRAM Microcontrollers, Fujitsu, 2009
http://www.fujitsu.com/downloads/EDG/binary/pdf/find/27-1e/1.pdf

[5] Ramtron International, WIKIPEDIA, 2012
http://en.wikipedia.org/wiki/Ramtron

[6] CamelForth, Bradford J. Rodriguez, 2009
http://www.camelforth.com/page.php?8

[7] MSP430F2012 MIXED SIGNAL MICROCONTROLLER, Texas Instr. Inc., 2011
http://www.ti.com/lit/ds/symlink/msp430f2012.pdf

[8] MSP430G2553, Texas Instruments Inc., 2012
http://www.ti.com/product/msp430g2553

[9] MSP430 LaunchPad Value Line Development kit, Texas Instruments Inc., 2012
http://www.ti.com/tool/msp-exp430g2

[10] MSP430 LaunchPad (MSP-EXP430G2), Texas Instruments Inc., 2012
http://processors.wiki.ti.com/index.php?title=MSP430_LaunchPad_(MSP-EXP430G2)

[11] Verslag Jaarvergadering Duitse Forth Vereniging, HCC!Forth, 2012
http://www.forth.hcc.nl/w/Nieuws/Nieuws

[12] MSP430 CamelForth version 0.3, B. J. Rodriguez, 2009
http://www.camelforth.com/download.php?view.12

[13] CamelForth/MSP430, B. J. Rodriguez, 2009
http://www.camelforth.com/page.php?8

[14] MSP430 Ultra-Low-Power MCUs, Texas Instruments Inc., 2002
http://www.ti.com/sc/docs/products/micro/msp430/msp430bulletin2.pdf

[15] CAMELFORTH FOR THE MSP430, Bradford J. Rodriguez, 2009
http://www.forth-ev.de/repos/4e4th/readme.430

[16] IAR Embedded Workbench Kickstart - Free 4KB IDE, Texas Instr. Inc., 2012
http://www.ti.com/tool/iar-kickstart&DCMP=MSP430&HQS=Other+OT+iarkickstart

[17] 4E4TH FOR THE MSP430G2553 on LaunchPad, Michael Kalus, 2012
http://www.forth-ev.de/repos/4e4th/README_4e4th.txt

[18] IAR Embedded Workbench for TI MSP430, Texas Instruments Inc., 2012
http://processors.wiki.ti.com/index.php/IAR_Embedded_Workbench_for_TI_MSP430

[19] Flash Programmers for TI's MSP430 MCU, Elpotronic Inc., 2012
http://www.elprotronic.com/download.html

[20] Rechnergesteuerte Funkalarmierung mit dem neuen FAS, Erich Bumiller, 1979
http://www.somersetweb.com/BruehlConsult/Projekte/FAS.html

[21] Notes on Structured Programming, Prof.dr. Edsger W. Dijkstra, 1970
http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF

[22] Threads of a FORTH Tapestry, Gregg Williams, BYTE, August 1980
http://www.colorforth.com/byte.htm

[23] RSC-FORTH User's Manual, Rockwell International, October 1983
http://www.smallestplcoftheworld.org/RSC-FORTH_User's_Manual.pdf

[24] The R65F11 and F68K Single-Chip Forth Computers, Randy M. Dumse,
The Journal of Forth Application and Research Volume2, Number 1, 1984
http://soton.mpeforth.com/flag/jfar/vol2/no1/article1.pdf

[25] Visual Basic, WIKIPEDIA 2012
http://en.wikipedia.org/wiki/Visual_Basic

[26] Win32Forth, Dirk Busch, 2007
http://win32forth.sourceforge.net/doc/p-index.htm

[27] Developmental History for ForthForm, Ezra Boyce, 2007
http://win32forth.sourceforge.net/doc/ForthForm/FF-History.htm

[28] visualFORTH, Dirk Bruehl, 2010
http://www.visualforth.blogspot.com/

[29] visualFORTH, Dirk Bruehl, 2010
http://www.visualforth.org/

[30] A picture is worth a thousand words, The Phrase Finder, 2012
http://www.phrases.org.uk/meanings/a-picture-is-worth-a-thousand-words.html

[31] Simpleware Principles, Dick Pountain, 2009
https://sites.google.com/site/dickpountainspages/home/simpleware

[32] Newbie Variants, WIKIPEDIA,2012
http://en.wikipedia.org/wiki/Newbie#Variants

[33] The Stellaris® ARM® Cortex™-M4F LaunchPad, Texas Instruments Inc., 2012
http://www.ti.com/ww/en/launchpad_site/stellaris.html?DCMP=stellaris-launchpad&HQS=stellaris-launchpad

[34] ThermoSense MK1, Andrew Reid, 2012
http://www.sustainabilitymeasurement.com

[35] TI Germany - Freising, Texas Instruments Inc., 2008
http://www.ti.com/europe/docs/sites/germany.htm

Notation Matters

Stephen Pelc
MicroProcessor Engineering
133 Hill Lane
Southampton SO15 5AF
England
t: +44 (0)23 8631 441
f: +44 (0)23 8033 9691
e: sfp@mpeforth.com
w: www.mpeforth.com

Abstract

Forth is an interactive extensible language. A consequence of this is that we can use the interactive nature of Forth both at compile-time and at run-time to produce our own notations. Good notation is very important when an application has a lifetime measured in decades.

During Forth Tagung 2012 at Beukenhof, I presented an improptu talk with this title. Michael Kalus then kept pestering me to write it up for *Vierte Dimensione*, the German Forth magazine. In the best impromptu talk tradition, I had kept no notes. Michael was good enough to transcribe the talk. Then I rewrote Michael's text to include examples and more text. And then I rewrote the paper again. Here it is.

The notation we use in Forth evolves over time. If we ever agree on a notation for OOP it will be a miracle. But, in order to achieve that, we have to remember some things about Forth that have nothing to do with **DUP** or **VARIABLE** but have a great deal to do with the way we use Forth. What we are using, what we are told Forth is really, is an interactive extensible language. And that means that we can change the notation of what we do. Which is actually one of the most powerful features of this class of language.

Using Forth at run-time

When I write a webserver in Forth, I don't have to write any scripting language. The server-side scripting in MPE's webservers has a mixture of HTML and scripting commands. First of all it tells us that it is going to use Forth as its scripting language:

```
<% language=forthscript %>
```

The **<%** and **%>** enclose scripting commands. I can include bits of Forth inside the web page to produce the output that I want, for example:
.

```
<% VAL @ . %>
```

That's an example of a script in Forth. We all know how to use it, and it cost almost nothing, either to implement or to use, and text interpretation happens at runtime. When I want to define how to open a COM port, I can write a peace of code that goes

```
COM1  9600 BAUD ... 8N1
```

and that is Forth source code. Even end-users can understand the code. What is important is to define a notation to perform the job. In both the examples above we are using the interactive capabilities of Forth at run-time, not just at compile-time. Quite often when we do this, we are providing a set of commands that an end-user needs to understand. For example, the people who write the web pages that use ForthScript need to know how to use it. We have to design the notation. These days, even embedded systems have enough memory to be able to do this.

The Forth community often tends to have an everlasting argument about the minutiae of the implementation, whereas what matters to users is the notation. This is the stuff that is on the paper or on the screen. When we design code for a Forth solution, design the notation before we design the implementation.

If the notation is clean to read, it is clean to use and we know we can write applications in it. Now, I am extremely biased. I am a vendor of Forth systems, which means I like to be paid. And the good thing about that is that I like big applications, because they cost a lot, and they become about writing maintaining lots of code, and that means when I come back to read the code in six month time, or in six years time, I still have to be able to understand what it does.

Application lifetime

You all know that you write comments on every line, and you still don't understand the code. The code should speak for itself. If you say something you just know what you mean. However, none of us are that good yet. You just hope that the person you are speaking to understands you. The maintenance programmer is the person you should be talking to when you write code. You don't know who he/she is or how clever he/she is.

I deal with applications that are more than 25 years old. The cross compiler that MPE sells is based upon code that we first saw in 1981. Very few lines of the original code remain unchanged, but the current code is a direct descendent of the original 1981 code. We have customers with code dating even further back. These people are dealing with code that is now over 30 years old, that started life on a HP desktop computer when it was the sexiest thing in computing. And we still have to maintain and extend those code bases.

What matters is that the code survives, and survives not in terms of its object code but in terms of its source code. The object code has gone from a 68000 to an 8086 to an 80386 to an ARM to something else. Every ten years it changes CPU or operating system. By the standards of the process, the application appears as source code on disc or paper. The the best form of preservation is what the developer sees.

Morse code application

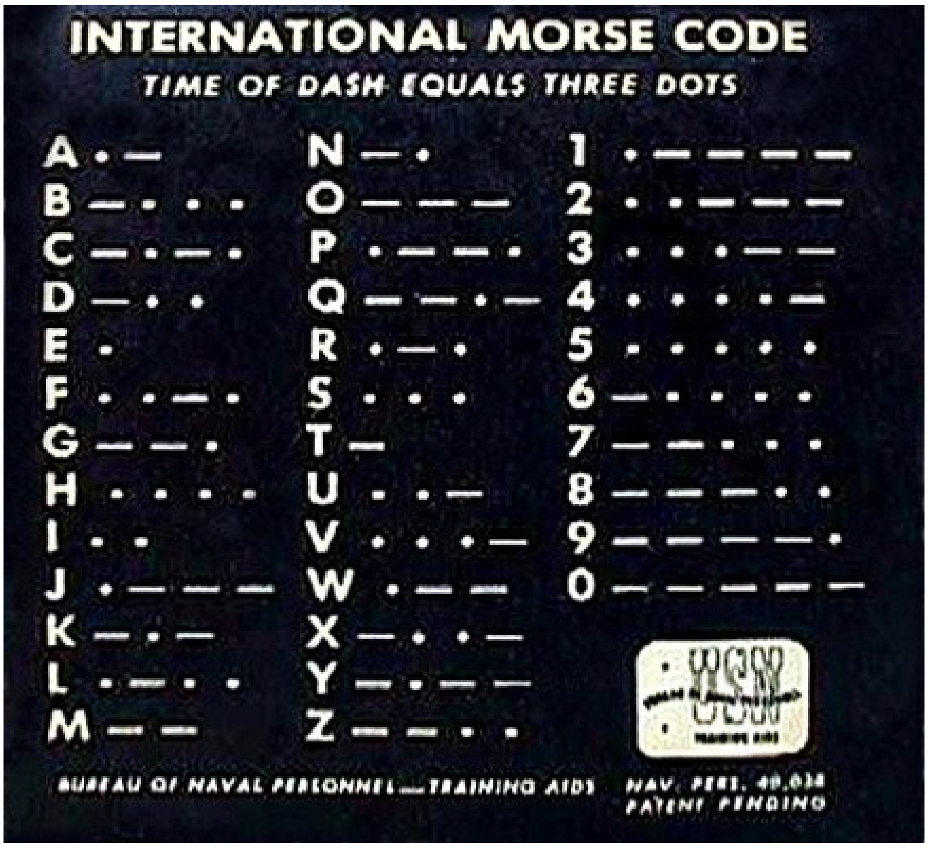
So when we come back to applications, for example our OOP package or to Carsten Strotman's jobs for Morse code, the question becomes how to define a notation for the package. I have chosen to look at entering Morse code into a Forth application. You could set up dots and dashes, where you can enter a dot/period, and a dash could be either an underline or a minus sign. Source tokens that contain just these letters must be Morse code characters. The characters are going to be printable character from the ASCII set. So we can say, for example

```
  morse-char A . -
  morse-char B - . . .
  morse-char C - . - .
```

or
```
  morse-char A .-
  morse-char B -...
  morse-char C -.-.
```

The second form avoids us having to know the end of the Morse sequence; it is just a space-delimited token containing dots and dashes.

We can translate this into some form that allows us to generate Morse code, either written in dots and dashes, or translated for transmission. The transmission side of the code will be responsible for knowing about inter-character and inter-word delays. For the moment, we are just interested in the notation, all the rest is implementation details. Our first job is to write the word **`MORSE-CHAR`** that generates one entry in our table. But that's not really enough, because the Morse code table has to be copied into the source code. And copying is really boring and really error-prone. What we want to do is to use existing text from a Morse code table. They typically look like these two.

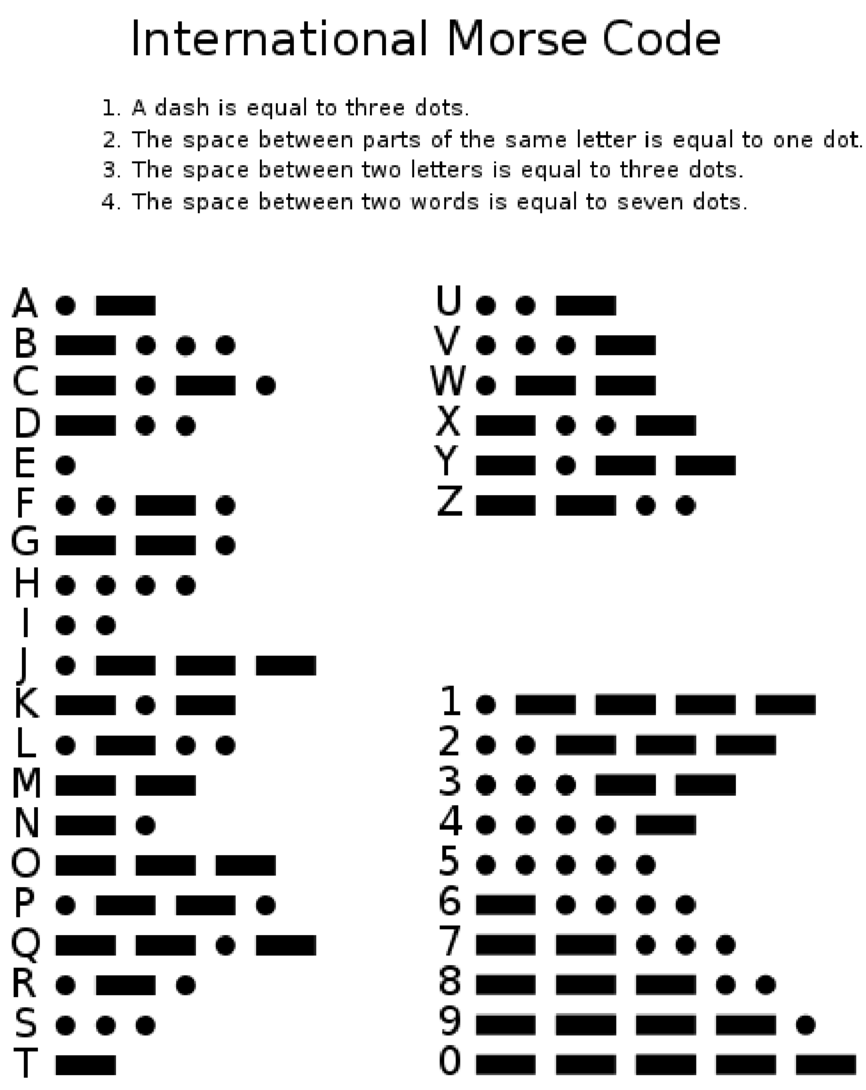


INTERNATIONAL MORSE CODE

TIME OF DASH EQUALS THREE DOTS

A	• —	N	— •	1	• — — — —
B	— • • •	O	— — —	2	• • — — —
C	— • — •	P	• — — •	3	• • • — —
D	— • •	Q	— — • —	4	• • • • —
E	•	R	• — •	5	• • • • •
F	• • — •	S	• • •	6	— • • • •
G	— — •	T	—	7	— — • • •
H	• • • •	U	• • —	8	— — — • •
I	• •	V	• • • —	9	— — — — •
J	• — — —	W	• — —	0	— — — — —
K	— • —	X	— • • —		
L	• — • •	Y	— • — —		
M	— —	Z	— — • •		

BUREAU OF NAVAL PERSONNEL — TRAINING AIDS NAV. PERS. 40,034 PATENT PENDING



International Morse Code

1. A dash is equal to three dots.
2. The space between parts of the same letter is equal to one dot.
3. The space between two letters is equal to three dots.
4. The space between two words is equal to seven dots.

A	● ▬	U	● ● ▬
B	▬ ● ● ●	V	● ● ● ▬
C	▬ ● ▬ ●	W	● ▬ ▬
D	▬ ● ●	X	▬ ● ● ▬
E	●	Y	▬ ● ▬ ▬
F	● ● ▬ ●	Z	▬ ▬ ● ●
G	▬ ▬ ●		
H	● ● ● ●		
I	● ●		
J	● ▬ ▬ ▬		
K	▬ ● ▬	1	● ▬ ▬ ▬ ▬
L	● ▬ ● ●	2	● ● ▬ ▬ ▬
M	▬ ▬	3	● ● ● ▬ ▬
N	▬ ●	4	● ● ● ● ▬
O	▬ ▬ ▬	5	● ● ● ● ●
P	● ▬ ▬ ●	6	▬ ● ● ● ●
Q	▬ ▬ ● ▬	7	▬ ▬ ● ● ●
R	● ▬ ●	8	▬ ▬ ▬ ● ●
S	● ● ●	9	▬ ▬ ▬ ▬ ●
T	▬	0	▬ ▬ ▬ ▬ ▬

Here is one in text form from
http://encyclopedia2.thefreedictionary.com/Morse+Code+%28table%29

```
International Morse Code

  A   .-        U   ..-
  B   -...      V   ...-
  C   -.-.      W   .--
  D   -..       X   -..-
  E   .         Y   -.--
                Z   --..
  F   ..-.
  G   --.       0   -----
  H   ....      1   .----
  I   ..        2   ..---
  J   .---      3   ...--
                4   ....-
  K   -.-       5   .....
  L   .-..      6   -....
  M   --        7   --...
  N   -.        8   ---..
  O   ---       9   ----.

  P   .--.      Period      .-.-.-
  Q   --.-      Comma       --..--
  R   .-.       ? Mark      ..--..
  S   ...       Hyphen      -....-
  T   -         Apostrophe  .----.
                Colon       ---...
  U   ..-       Quotation   .-..-.
  V   ...-      Slash       -..-.
  W   .--       @ sign      .--.-.
  X   -..-
  Y   -.--
  Z   --..
```

This is a good starting point for our Morse code table, so let us see what we have to change in the the table. We know that its going be a single character is followed by a sequence of dots and dashes. Looking at the table above, we see that we are looking for pairs of tokens. The first is the character and the second is its representation in dots and dashes. This is handled by the word **MORSE-CHAR** that we defined earlier. The effort of changing "Period" to "." is acceptable.

With a bit of syntactic sugar, we can generate a complete Morse code table that is easy to maintain. None of this is clever, it is all about achieving a result with the least effort, which in turn is just good engineering practice. The key decision is the decision to be lazy.

```
create MorseTable  \ -- addr
[morse
  A   .-       U   ..-
  B   -...     V   ...-
  C   -.-.     W   .--
  D   -..      X   -..-
  E   .        Y   -.--
               Z   --..
  F   ..-.
  G   --.      0   -----
  H   ....     1   .----
  I   ..       2   ..---
  J   .---     3   ...--
               4   ....-
  K   -.-      5   .....
  L   .-..     6   -....
  M   --       7   --...
  N   -.       8   ---..
  O   ---      9   ----.

  P   .--.     .   .-.-.-
  Q   --.-     ,   --..--
  R   .-.      ?   ..--..
  S   ...      -   -....-
  T   -        '   .----.
               :   ---...
  U   ..-      "   .-..-.
  V   ...-     /   -..-.
  W   .--      @   .--.-.
  X   -..-
  Y   -.--
  Z   --..
morse]
```

Using a good notation leads to reliability. Think about writing a 256 character font table. You are going to see an ASCII character followed by rows of bit on/off definitions. When you first see this data your eyes will glaze over at the potential boredom. If you think that you are going to translate this stuff to hexadecimal bytes without any errors, then think again. You will make mistakes. So the question is, should you spend a short time to write a notation for this and future font tables? Or should you spend a long time debugging?

Conclusion

What matters is notation. We are dealing with Forth which is a language in which we can manipulate our notation. We should remind ourselves to take advantage of this feature.

Acknowledgements

Elizabeth Rather never lets a notation go unchallenged.

Michael Kalus made me take this paper seriously.

Forth Semantics for Compiler Verification

Bill Stoddart, Campbell Ritchie, Steve Dunne

September 13, 2012

Abstract

Here we are interested in the semantics of Forth from the point of view of using Forth as a target language for a formally verified compiler for Ruth-R, a reversible sequential programming language we are currently developing. We limit out attention to those Forth operations and constructs which will be targetted by the Ruth-R compiler. To facilitate the comparison of meanings of source and target languages, we represent the semantics of Forth code by translation into a form which can be described using the "prospective value" semantics we use for Ruth-R.

1 Introduction

We are interested in the semantics of Forth from the point of view of using Forth as a target language for a formally verified compiler.

Our source language for this project is Ruth-R, an expressive reversible guarded command language for which we are currently constructing the syntax, semantics, and compiler. The semantics of Ruth-R are expressed in terms of the value some expression E can take after the execution of some program S, which we represent as $S \diamond E$. We call this the prospective value of E after executing S, and refer to the semantics as "prospective value semantics", or PV semantics for short.

The target language is RVM-Forth [11], a reversible version of Forth we have developed to explore the algorithmic possibilities of reversible languages.

Our approach will be to give a translation of each Forth operation into a form to which PV semantics may be applied. That is, we translate it into the form of a sequential programming language, with the stack appearing as a variable.

Our investigations also cover different representations of the same value. For example, RVM-Forth provides a generic set package, but for efficiency reasons we may wish to represent small sets using bit vectors.

A key design question for us will be how to model the Forth parameter stack, and we will compare two possible approaches: first as a sequence of cell values, (an approach previously investigated by P Knaggs[4]) whose entries are subsequently viewed in terms of interpretation functions; and secondly as a "conceptual stack", a structure which is directly able to hold values of any type.

Our approach is also influenced by the fine grained nature of Forth semantics; the infix language assignment $x := x + 1$ will compile to Forth as `x 1 + to x`; and for compositionality we require separate semantic translations for each of `x, 1, +` and `to x` and the sequential composition of these to obtain the semantics of `x 1 + to x` . This makes it essential that the semantic description of the sequential composition of operations be expressed in a simple way.

Our approach is more detailed than the Pöial algebra of stack effects [7] but is less ambitious that some other previous formulations. We do not seek a technique for the complete description for the Forth virtual machine[6] or the operation of the Forth compiler [5], but we rather extend Forth with the data types and control structures required by Ruth-R, and then give semantic descriptions of these components; thus the work described here can be characterised as a shallow formulation. Its advantages are that is is axiomatic, fully compositional, and minimises the semantic distance between source and target languages.

Our general approach is conditioned by that we have adopted for expressing the semantics of reversible language. We borrow many techniques from the B-Method [1] as described in [12], and to simplify our theory presentation we use Hehner's conception of a "bunch" as the contents of a set.

The rest of the paper is structured as follows: section 2 deals with mathematical background and the axioms of our semantics; section 3 sketches an algebraic treatment of stack semantics; Section 4 considers two different models for the stack, and discusses assignment, literal values, and differing representations of the same data; section 5 defines the semantics of selection and iteration; section 6 discusses local variables; in section 7 we conclude.

2 Mathematical Preliminaries

2.1 Bunch Theory

Following Hehner[2], we give a mathematical meaning to the contents of a set, which we call a bunch: e.g. the contents of the set $\{1, 2\}$ is the bunch $1, 2$. We write $\sim A$ for the contents of set A, thus $\sim\{1, 2\} = 1, 2$.

The comma used in a bunch extension expression such as $1, 2, 5$ is now a mathematical operator, called bunch union. It is associative and commutative, and its precedence is just below than that of the expression connectives. It is associated with set union through the rule

$A \cup B = \{\sim A, \sim B\}$

Binary operations applied to bunches are lifted to apply pairwise, thus adding the bunches $0, 1$ and $2, 4$ yields $0+2, 0+4, 1+2, 1+4$. Note that we cannot write this sum as $0, 1 + 2, 4$ because bunch union (comma) has a lower precedence than +; nor can we use standard brackets to enforce precedence, as we retain the use of brackets to express tuples. We can however bracket with $\sim\{..\}$, as in $\sim\{0,1\} + \sim\{2,4\} \;=\; 2,4,3,5$

A bunch A is a sub-bunch of B if each element of A is an element of B. We write this as $A \,:\, B$. Sub-bunches are related to subsets by the rule:

$A \,:\, B \;\Leftrightarrow\; \{A\} \subseteq \{B\}$

The guarded bunch $p \longrightarrow E$ is equal to the empty bunch, *null*, if predicate p is false, and otherwise equal to E. The preconditioned bunch $p \;|\; E$ is equal to the improper bunch $\perp$ if p is false, and otherwise equal to E. The improper bunch expresses complete ignorance about a value, extending to a suspicion that a computation supposed to produce a value might have crashed.

We use a typed set theory based on the axiomatic approach used in the B-Method. We assume the availability of any required given sets, including the integers, and we are able to produce from these new maximal sets by the use of set product and powerset operations. These maximal sets act as types. We can generate new sets by set comprehensions of the form

$\{x \mid x \in X \wedge P \bullet E\}$

where expression E is of a fixed type, so that our set comprehensions can only produce homogeneous sets.

We similarly restrict our attention to bunches which are homogeneous; these have the same type as their elements, thus $1,2$ is of type integer, which we can write as $1,2 \in \mathbb{Z}$.

Analogously to set comprehension, we can write the bunch comprehension:

$\oint x \bullet E$

Here, although the type of x is not given explicitly it has to be implicit from an examination of the expression E. The result consists of all values that E can take as x ranges over its type. For example the bunch $10,20$ can be written as the bunch comprehension $\oint x \bullet x : 1,2 \longrightarrow 10 * x$.

2.2 Prospective Value Semantics

We are formalising a reversible language with a choice construct used to express both non-determinism and provisional choices subject to backtracking. We write $S \diamond E$ for the values expression E could take were program S to be executed. We have the following rules which define $S \diamond E$ over the essential semantic components of the language:

Name	Rule	Condition
skip	$skip \diamond E \;=\; E$	
assignment	$x := F \diamond E \;=\; (\lambda\, x \bullet E)F$	
pre-condition	$P \mid S \diamond E \;=\; P \;\pmb{\mid}\; (S \diamond E)$	
choice	$S \;[\!]\; T \diamond E \;=\; (S \diamond E)\,,\,(T \diamond E)$	
guard	$P \xrightarrow{\Box} S \diamond E \;=\; P \longrightarrow (S \diamond E)$	
seq comp	$S \,;\, T \diamond E \;=\; S \diamond T \diamond E$	
local variable	$var\, x \,.\, S \diamond E \;=\; \oint x \bullet S \diamond E$	x not free in E

The precedence of $\diamond$ is below that of programming connectives, whose precedence, in descending order, is $:=\,,\ \xrightarrow{\Box}\,,\ [\!]\,,\ ;\,,\ \mid$ The large equals $=$ has the same meaning as "=" but a very low precedence: we require it when discussing equality in the context of programs as the precedence of the standard equal sign is above that of the program connectives.

Our use of bunches enables us to express the effects of choice and sequential composition in a homogeneous manner, and to describe non-determinism. This case is more fully argued in [8].

The attentive reader will note that these semantic components do not include selection and iteration constructs. These are handled by means of choice and guard. e.g

if g **then** S **else** T **end**

is expressed as:

$g \xrightarrow{\Box} S \;[\!]\; \neg\, g \xrightarrow{\Box} T$

This decomposition was first proposed by Hehner [2, 3] who used predicative semantics. Abrial adapted it to predicate transformer semantics for use in in the B-Method [1]. A description of our approach using prospective value semantics is given in [10]. We can extend this to probabilistic programs [8], and express preference within provisional choice [9].

3 Prospective values and stack algebra

We will consider two different ways a stack may be modelled in terms of an underlying representation within our mathematical world of typed set theory, before choosing our preferred representation. However, we will first deal *algebraically* with some basic stack manipulations. This will avoid some repetition, as the algebraic treatment will be the same for either model.

We represent an empty stack by ε. If s is a stack state, let $s \smile x$ be the new stack state obtained by pushing x.

If s is a non-empty stack, $drop(s)$ will be the new stack obtained by dropping the top item. We thus have

$drop(s \smile x) \;=\; s$

Let $top(s)$ be the top element of a non-empty stack s, and $next(s)$ the second from top element of a stack s which has at least two elements. Thus:

$top(s \mathbin{\llcorner\!\lrcorner} x) \;=\; x$ and $next(s \mathbin{\llcorner\!\lrcorner} x \mathbin{\llcorner\!\lrcorner} y) \;=\; x$

The function *swap* takes a stack and returns the new stack obtained by swapping the top two elements. Thus

$swap(s \mathbin{\llcorner\!\lrcorner} x \mathbin{\llcorner\!\lrcorner} y) \;=\; s \mathbin{\llcorner\!\lrcorner} y \mathbin{\llcorner\!\lrcorner} x$

We define the semantics of the Forth `SWAP` operation, which unlike the function *swap*, acts on a particular stack (which we just call *stack*):

$[\![\texttt{SWAP}]\!]^{\mathcal{F}} \;=\; depth(stack) \geq 2 \mid stack := swap(stack)$

Here, the notation $[\![\texttt{SWAP}]\!]^{\mathcal{F}}$ encloses the Forth code being discussed in the semantic brackets $[\![\ldots]\!]^{\mathcal{F}}$. In general, if `S` is Forth code, $[\![\texttt{S}]\!]^{\mathcal{F}}$ will represent its translation into a form whose meaning can be expressed in PV semantics.

With the above semantic definition of `SWAP` we bring a Forth stack manipulation within the scope of PV semantics by treating it as an assignment. The semantics of `SWAP` given above also tells us that its frame (the list of variables it may alter) contains just *stack*. Also, by PV rules for pre-condition and assignment:

$[\![\texttt{SWAP}]\!]^{\mathcal{F}} \diamond stack \;=\; depth(stack) \;\geq\; 2 \;\blacktriangleright\; swap(stack)$

We can apply similar treatments to other stack manipulation operations.

$[\![\texttt{DROP}]\!]^{\mathcal{F}} \;=\; depth(stack) \;\geq\; 1 \mid stack := drop(stack)$
and hence
$[\![\texttt{DROP}]\!]^{\mathcal{F}} \diamond stack \;=\; depth(stack) \;\geq\; 1 \mid drop(stack)$

For `NIP` we introduce an auxiliary function *nip* such that for any stack s and items x, y we have $nip(s \mathbin{\llcorner\!\lrcorner} x \mathbin{\llcorner\!\lrcorner} y) \;=\; s \mathbin{\llcorner\!\lrcorner} y$, allowing us to define the semantics of the Forth operation `NIP` as:

$[\![\texttt{NIP}]\!]^{\mathcal{F}} \;=\; stack := depth(stack) \;\geq\; 2 \;\blacktriangleright\; stack := nip(stack)$

and hence

$[\![\texttt{NIP}]\!]^{\mathcal{F}} \diamond stack \;=\; depth(stack) \;\geq\; 2 \;\blacktriangleright\; nip(stack)$

In addition to giving the meaning of individual Forth operations, we also need to express the meaning of Forth's program connectives. The first we require is sequential composition:

$[\![\texttt{S T}]\!]^{\mathcal{F}} \;=\; [\![\texttt{S}]\!]^{\mathcal{F}} \,;\, [\![\texttt{T}]\!]^{\mathcal{F}}$

We are now in a position to prove a semantic equality. We will show that `NIP` is equivalent to `SWAP DROP`. We define two Forth programs to be equal if their semantic representations are equal. i.e. for Forth programs A and B:

$A \;=\; B \;\widehat{=}\; [\![\texttt{A}]\!]^{\mathcal{F}} \;=\; [\![\texttt{B}]\!]^{\mathcal{F}}$

and we define their semantic representation to be equal if they have the same frame and the same PV effect over the variables of that frame (or, equivalently, the same PV effect over an arbitrary expression).

We will show that: $[\![\texttt{SWAP DROP}]\!]^{\mathcal{F}} \;=\; [\![\texttt{NIP}]\!]^{\mathcal{F}}$

Since the frame of $[\![\texttt{SWAP DROP}]\!]^{\mathcal{F}}$ and the frame of $[\![\texttt{NIP}]\!]^{\mathcal{F}}$ are both *stack*, we can show the required equality by showing

$[\![\texttt{SWAP DROP}]\!]^{\mathcal{F}} \diamond \mathit{stack} \;=\; [\![\texttt{NIP}]\!]^{\mathcal{F}} \diamond \mathit{stack}$

$[\![\texttt{SWAP DROP}]\!]^{\mathcal{F}} \diamond \mathit{stack}$

$=$ Forth seq comp

$[\![\texttt{SWAP}]\!]^{\mathcal{F}} \,;\, [\![\texttt{DROP}]\!]^{\mathcal{F}} \diamond \mathit{stack}$

$=$ PV seq comp

$[\![\texttt{SWAP}]\!]^{\mathcal{F}} \diamond [\![\texttt{DROP}]\!]^{\mathcal{F}} \diamond \mathit{stack}$

$=$ semantics of *DROP*

$[\![\texttt{SWAP}]\!]^{\mathcal{F}} \diamond \mathit{depth}(\mathit{stack}) \geq 1 \mid \mathit{stack} := \mathit{drop}(\mathit{stack})$
$\diamond \mathit{stack} \;=$ PV pre-cond

$[\![\texttt{SWAP}]\!]^{\mathcal{F}} \diamond \mathit{depth}(\mathit{stack}) \geq 1 \;\pmb{|}\; \mathit{stack} := \mathit{drop}(\mathit{stack})$
$\diamond \mathit{stack} \;=$ PV assignment

$[\![\texttt{SWAP}]\!]^{\mathcal{F}} \diamond \mathit{depth}(\mathit{stack}) \geq 1 \;\pmb{|}\; \mathit{drop}(\mathit{stack})$

$=$ semantics of *SWAP*

$\mathit{depth}(\mathit{stack}) \geq 2 \mid \mathit{stack} := \mathit{swap}(\mathit{stack})$
$\diamond \mathit{depth}(\mathit{stack}) \geq 1 \;\pmb{|}\; \mathit{drop}(\mathit{stack})$

$=$ PV pre-condition

$\mathit{depth}(\mathit{stack}) \geq 2 \;\pmb{|}\; (\mathit{stack} := \mathit{swap}(\mathit{stack})$
$\diamond \mathit{depth}(\mathit{stack}) \geq 1 \;\pmb{|}\; \mathit{drop}(\mathit{stack}))$

$=$ by assignment

$\mathit{depth}(\mathit{stack}) \geq 2 \;\pmb{|}\; \mathit{depth}(\mathit{swap}(\mathit{stack})) \geq 1$
$\pmb{|}\; \mathit{drop}(\mathit{swap}(\mathit{stack}))$

$=$ simplifying pre-cond

$\mathit{depth}(\mathit{stack}) \geq 2 \;\pmb{|}\; \mathit{drop}(\mathit{swap}(\mathit{stack}))$

Now for $\mathit{depth}(\mathit{stack}) \geq 2$ there will be some stack s and items x, y such that $\mathit{stack} \,=\, s \text{\textvisiblespace} x \text{\textvisiblespace} y$ and hence:

$[\![\texttt{SWAP DROP}]\!]^{\mathcal{F}} \diamond \mathit{stack} \;=$

$\mathit{depth}(\mathit{stack}) \geq 2 \;\pmb{|}\; \mathit{drop}(\mathit{swap}(s \text{\textvisiblespace} x \text{\textvisiblespace} y) \;=$

applying function *swap*

$\mathit{depth}(\mathit{stack}) \geq 2 \;\pmb{|}\; \mathit{drop}(s \text{\textvisiblespace} y \text{\textvisiblespace} x) \;=$

applying function *drop*

$\mathit{depth}(\mathit{stack}) \geq 2 \;\pmb{|}\; s \text{\textvisiblespace} y \;=$

from property of *nip*

$\mathit{depth}(\mathit{stack}) \geq 2 \;\pmb{|}\; \mathit{nip}(s \text{\textvisiblespace} x \text{\textvisiblespace} y) \;=$

equality: $\mathit{stack} \,=\, s \text{\textvisiblespace} x \text{\textvisiblespace} y$

$\mathit{depth}(\mathit{stack}) \geq 2 \;\pmb{|}\; \mathit{nip}(\mathit{stack}) \;=$

assignment introduction

$\mathit{depth}(\mathit{stack}) \geq 2 \mid \mathit{stack} := \mathit{nip}(\mathit{stack})$
$\diamond \mathit{stack} \;=$

semantics of *NIP*

$[\![\texttt{NIP}]\!]^{\mathcal{F}} \diamond \mathit{stack}$

4 Modelling the stack

Modelling a stack which may contain items of any type is a challenge in our semantics, where we rely on a typed set theory which only admits homogeneous sets. We consider two possible approaches. In the first the stack is modelled as a sequence of raw cell values, and we use interpretation functions to tell us what is represented by these cells. In the second we model a stack state as an n-tuple, taking advantage of the fact that different elements of a tuple need not be homogeneous, and we describe a conceptual stack of possibly non-homogeneous mathematical objects.

4.1 Modelling the stack as a sequence of cells

In this section we introduce a stack modelling technique which we will subsequently reject in favour of a more abstract approach. The section is included mainly to show some difficulties that arise from this approach, and can be omitted without affecting the readers understanding of the rest of the paper.

We might model the stack as a finite sequence of cells, where a cell is a function defined on 32 bit machines as:

$CELL = 0..31 \nrightarrow BIT$

A stack is a finite sequence of cells. We introduce a constant, the set of all stacks, which we call $STACK$.

$STACK = fseq(CELL)$

and we introduce a variable *stack* to represent the parameter stack, with the invariant property:

$stack \in STACK$

With this model, pushing an element onto a stack is simply a matter of appending an element to a sequence:

$s \in STACK \wedge c \in CELL \Rightarrow s \llcorner c = s \frown \langle c \rangle$

Functions which perform calculations on a stack will interpret its cell values in a particular way. For example the operation $+$ might interpret them as signed numbers. Let $num \in CELL \rightarrow -2^{31} .. (2^{31} - 1)$ be the "interpretation function" giving the signed integer value associated with the cell under a two's complement representation. The inverse of *num* is also a function, such that for any signed integer n in the representable range, $num^{-1}(n)$ will be the cell that represents n.

To describe the semantics of the Forth word $+$, we first define a function *plus* which acts on any stack of depth ≥ 2 and adds the top two elements: we see in its definition the explicit interpretation of cell values as numbers:

$plus \in STACK \nrightarrow STACK$
$dom(plus) = \{s \mid s \in STACK \wedge card(s) > 1\}$
$s \in STACK \wedge c_1, c_2 : CELL \Rightarrow$
$\quad plus(s \frown \langle c_1, c_2 \rangle) = s \frown \langle num^{-1}(num(c_1) + num(c_2)) \rangle$

The attentive reader will not that, in defining *plus*, we have not been careful about what happens if the application of + yields a value outside the expressible range. This is because we avoid any responsibility for the the value provided by such an application by including an appropriate clause in the pre-condition in the semantics of the Forth operation +:

$[\![+]\!]^{\mathcal{F}} = \quad num(next(stack)) + num(top(stack)) \in ran(num) \wedge depth(stack) \geq 2$
$\mid \; stack := plus(stack)$

The above example shows an approach we can take when the interpretation functions we are using are constant. However, for some operations the results will be data structures held in the heap and referenced by pointers on the stack. Consider, for example, the following expression in RVM-Forth which represents $\{100, 200\} \cup \{300\}$:

```
INT { 100 , 200 , }  INT { 300 , }  ∪
```

As each set is evaluated, we obtain, on the stack, a pointer to a structure on the heap. Let *set* be the interpretation function that maps the cell values of these pointers to their corresponding sets. We note that *set* is a variable which changes with every new set that is produced. Also, the pointer values are not explicitly defined: we can only say that some suitable pointer is provided, which we must do by introducing it with an existential quantifier. One possible description of $[\![\cup]\!]^{\mathcal{F}}$ is:

$depth(stack) \geq 2 \wedge top(stack) \in dom(set) \wedge next(stack) \in dom(set) \Rightarrow$
$\exists \; c \bullet [\![\cup]\!]^{\mathcal{F}} =$
$stack := drop(drop(stack)) \smile c \quad \|$
$set := set \cup \{ (c, \; set(next(stack) \cup set(top(stack)) \}$

The reader will note that instead of an explicit definition of the semantics of set union, we have obtained an implicit description in which the term $[\![\cup]\!]^{\mathcal{F}}$ has become a fully fledged mathematical object. This is not a situation we relish: it imposes some additional responsibilities to show the mathematical validity of such terms, and does not provide for equational reasoning.

These concerns, along with the entailment of irrelevant details concerning heap pointers, motivate us to investigate an approach in which we model a non-homogeneous stack of conceptual mathematical objects.

4.2 Modelling the conceptual stack

We use a typed set theory whose axioms are given in [1]. We can define some "given sets" and from these we form new sets by means of the powerset and set product operations. Set comprehension allows us to describe subsets of these given and constructed sets.

We wish to model a "conceptual stack", which may include different types of mathematical object. The restrictions of our type theory rule out the use of a sequence as the modelling vehicle, so we turn to tuples.

In a formalism which provides n-tuples, we could model a stack containing $a\ b\ c$ with the 3-tuple (a, b, c). However, our formalism provides only ordered pairs, and when we write (a, b, c) we obtain an ordered pair $(\,(a, b),\ c\,)$.

Thus we cannot just use (a, b, c) to model the stack containing a, b and c: since ordered pairs are themselves a possible stack item, there would be no way to distinguish a 3 item stack containing a, b and c from a two item stack containing (a, b) and c (we note that tuple construction is left associative). Furthermore, we would have no way to represent a stack of 0 or 1 items.

We can, however, construct some special ordered pairs which will model n-tuples, or stacks. We require an arbitrary constant value to represent an empty stack, and we call this ε.

For a stack s we define the act of pushing an element s using tuple construction as:

$s \mathbin{\text{\textvisiblespace}} x \ \widehat{=}\ (s, x)$

We make use of the ordered pair notations:

$\mathit{first}(x, y)\ \widehat{=}\ x, \quad \mathit{second}(x, y)\ \widehat{=}\ y$

And thus we can define the function that yields the top element of a stack:

$\mathit{top}(s)\ \widehat{=}\ \mathit{second}(s)$

If the function *top* is applied to an empty stack we obtain, in our particular theory of partial function application, the empty bunch. However, we will always impose pre-conditions on operations in order to wash our hands of any responsibility for such an application.

We also have:

$\mathit{drop}(s)\ \widehat{=}\ \mathit{first}(s)$
$\mathit{next}(s)\ \widehat{=}\ \mathit{top}(\mathit{drop}(s))$
$\mathit{depth}(s)\ \widehat{=}\ s = \varepsilon \longrightarrow 0,\ s \neq \varepsilon \longrightarrow \mathit{depth}(\mathit{drop}(s)) + 1$

At this point the diligent reader may wish to construct the stack s where $s = \varepsilon \mathbin{\text{\textvisiblespace}} a \mathbin{\text{\textvisiblespace}} b \mathbin{\text{\textvisiblespace}} c$ and evaluate $\mathit{top}(s)$, $\mathit{drop}(s)$, $\mathit{next}(s)$ and $\mathit{depth}(s)$.

Since we cannot characterise s as a stack by saying it is a member of some type (which for us is a maximal set) we introduce a unary predicate *IsStack*, with the following defining properties:

$\mathit{IsStack}(\varepsilon)$
$\mathit{IsStack}(s) \Rightarrow \mathit{IsStack}(s \mathbin{\text{\textvisiblespace}} x)$
and nothing else is a stack.

We now return to the semantics of set union which gave us some trouble when modelling the stack as a sequence of cells. We begin, as usual, by the description of an auxiliary function.

$IsStack(s) \wedge \exists\ T \bullet x \subseteq T \wedge y \subseteq T \Rightarrow union(s \frown x \frown y) = s \frown x \cup y$

Then we can give an explicit definition of the semantics of Forth set union:

$[\![\cup]\!]^{\mathcal{F}} = \exists\ T \bullet top(stack) \subseteq T \wedge next(stack) \subseteq T \mid$
$stack := union(stack)$

From now on we take the conceptual stack as our model.

4.3 On differing representations of the same data

Within a computer system we may have different representations of the same data. A string could be a counted string or an ascii zero string. A sequence of n elements has a representation in the RVM sets package in terms of its graph, i.e. as a set of ordered pairs, but could also be represented by an n element array. A function could be represented as an operation or, in passive form, by its graph, or, if it is a sequence, as an array.

To show how such variations in representation may be handled using the conceptual stack we consider the example of bitsets, which can represent subsets of 0..31 according the the bit settings in a 32 bit cell.

We define a function *bits2set* whose domain is the subsets of 0..31 and which maps its argument to the corresponding bitset representation for that set. The semantics of the corresponding Forth operation, `Bits2Set` is:

$[\![\texttt{Bits2Set}]\!]^{\mathcal{F}} = depth(stack) > 0 \wedge top(stack) \subseteq 0..31 \mid$
$stack := drop(stack) \frown bits2set(top(stack))$

4.4 Literal values, variables, and assignments

Some RVM-Forth literal expressions are written in non-standard notation. For example the set $\{10, 20\}$ would be written in RVM-Forth as `INT { 10 , 20 , }`. In the following we use $[\![\texttt{L}]\!]^{\mathcal{L}}$ to represent the translation of the RVM-Forth literal expression `L` into standard notation, but we do not give this translation in detail.

We express the semantics of a literal expression `L` as:

$[\![\texttt{L}]\!]^{\mathcal{F}} = stack := stack \frown [\![\texttt{L}]\!]^{\mathcal{L}}$

Let `x` be a variable (i.e. a Forth `VALUE`). We assume the existence of a corresponding variable in the mathematical world, which we write, in mathematical typeface, as x. The semantic representations for `x` are:

$[\![\texttt{x}]\!]^{\mathcal{F}} = stack := stack \frown x$

$[\![\texttt{to x}]\!]^{\mathcal{F}} = depth(stack) > 0 \mid stack := drop(stack) \parallel x := top(stack)$

5 Selection and Iteration

Instead of a condition test which is syntactically connected to the selection statement, the Forth `IF` structure uses the value at the top stack item, assumed to hold the result of a previous test.

$\llbracket\ \textbf{if}\ \texttt{S}\ \textbf{else}\ \texttt{T}\ \textbf{then}\ \rrbracket^{\mathcal{F}}\ =$
$\mathit{var}\,\tau\,.\,\tau\ =\ \mathit{top}(\mathit{stack})\ \xrightarrow{\square}\ \mathit{stack} := \mathit{drop}(\mathit{stack})\ ;$
$\ \textbf{if}\ \tau \neq 0\ \textbf{then}\ \llbracket\texttt{S}\rrbracket^{\mathcal{F}}\ \textbf{else}\ \llbracket\texttt{T}\rrbracket^{\mathcal{F}}\ \textbf{end}$

For iteration, we restrict our attention to the structure:

$\llbracket\texttt{BEGIN G WHILE S REPEAT}\rrbracket^{\mathcal{F}}$

since this most closely resembles the classic while loop representation used in the formal semantics of sequential programming; to obtain the classical loop interpretation we must impose the following restrictions:

$\mathit{frame}(\llbracket\texttt{G}\rrbracket^{\mathcal{F}})\ =\ \mathit{stack}$

$\llbracket\texttt{G}\rrbracket^{\mathcal{F}} \diamond \mathit{stack}\ =\ \ \mathit{stack} \mathbin{\text{␣}} E\quad \text{for some } E$

$\llbracket\texttt{S}\rrbracket^{\mathcal{F}} \diamond \mathit{stack}\ =\ \ \mathit{stack}$

these assumptions allow us to transcribe the loop into a standard sequential programming representation $\llbracket\texttt{BEGIN G WHILE S REPEAT}\rrbracket^{\mathcal{F}}\ =$
$\textbf{while}\ \mathit{top}(\llbracket\texttt{G}\rrbracket^{\mathcal{F}} \diamond \mathit{stack}) \neq 0\ \textbf{do}\ \llbracket\texttt{S}\rrbracket^{\mathcal{F}}\ \textbf{end}$

6 Local variables

Local variables in RVM-Forth operations are used to capture operation arguments and other local variables whose initial values are provided after execution of an operation has begun. An example of local variable syntax in RVM-Forth is:

```
: T (: x y :) 100 VALUE u  200 VALUE v  ...  ;
```

Here `x` and `y` will be initialised with the value of the next and top stack items when `T` is invoked, and `u` and `v` will be initialised with the values 100 and 200.

Our semantics of local variables will be described in terms of initial values which are taken from the Forth parameter stack. However, we wish to accommodate an implementation technique which leaves the values where they are, and accesses them by indexing into the Forth parameter stack. This provides more efficient code on register based architectures, but we need to follow two rules to ensure correct usage.

The first is that an operation whose arguments are instantiated as local variables must not access any stack values held below these arguments. Here is an example of declaration of local parameters that breaks that rule:

```
: SURFA ( a:n b:n c:n -- 2*(a*b + a*c + b*c)) (: b c :) ...
```

here, the value supplied for argument `a` needs to be accessed from the stack, but due to our implementation technique of leaving local values on the parameter stack, it will be hidden below the values supplied for `b` and `c`. To allow `a` to be accessed as a local variable we could begin our definition with:

```
: SURFA ( a:n b:n c:n -- 2*(a*b + a*c + b*c)) (: a b c :) ...
```

The second rule is that the code between the parameter list and any instance of `VALUE` must have a stack signature of the form `( -- x )`. In terms of our Forth semantics this means that, where `S` represents the code between the end of the parameter stack and the relevant occurrence of `VALUE`, then for some expression E:

$$[\![\texttt{S}]\!]^{\mathcal{F}} \diamond \mathit{stack} \;=\; \mathit{stack} \,\llcorner\, E$$

Two illustrations will serve as justification: first we modify a previous example so it breaks the rule:

```
 : T (: x y :) 100 200 VALUE u VALUE v  .. ;
```

We now have code between the end of the parameter list and an occurrence of `VALUE` which adds two items to the stack. According to the semantics we will give, this will initialise `u` to 200 and `v` to 100, but in our implementation, which leaves local variables on the stack, it will still initialise `u` to 100 and `v` to 200. This happens as follows: the Forth compilation of `(: x y :)` sets up the top two stack cells to be a stack frame for variables `x` and `y`. The declarations `VALUE u` and `VALUE v` each extend the stack frame by one cell, associating with these cells the names `u` and `v`. Thus when 100 and 200 appear on the stack at run time, they provide the initial values for `u` and `v` respectively. Our rule ensures the synchronisation of compile time and run time activity.

A second way in which the rule can be broken is illustrated by the following code:

```
: T (: x :) DUP VALUE y .. ;
```

Here the programmer is making use of his knowledge of an implementation which leaves local variables on the parameter stack. Where this is the case, this example will initialise `y` with the same value as `x`. However, this is not formally correct as our semantics will insist that the initial value for `x` has been removed from the stack.

6.1 On Scope

The semantics of local variables requires an idea of the following code, indicated by `S` in the rules below, within which the local variable is in scope. This scope is, by default, till the end of the Forth definition in which the local is declared, and otherwise is controlled by the use of scoping brackets.

6.2 Formal semantics of locals

The last local declared in the list of operation parameters is initialised from the top of stack

$[\![(\texttt{:}\ \ \ldots\ \texttt{x}\ \ \texttt{:})\ \texttt{S}]\!]^{\mathcal{F}} \ =$
$var\, x\, .\, x = top(stack) \xrightarrow{\square} stack := drop(stack)\ ;\ [\![(\texttt{:}\ \ \ldots\ \ \texttt{:})\ \texttt{S}]\!]^{\mathcal{F}}$

This above rule is applied until we obtain an empty parameter list, which has the following empty frame semantics:

$[\![(\texttt{:}\ \texttt{:})\ \texttt{S}]\!]^{\mathcal{F}} \ = \ [\![\texttt{S}]\!]^{\mathcal{F}}$

Local variables declared following the operation's parameter list are formally initialised from the stack:

$[\![\texttt{VALUE}\,\texttt{x}]\!]^{\mathcal{F}} \ = \quad var\, x\, .\, x = top(stack) \xrightarrow{\square} stack := drop(stack)$

7 Conclusions

We have chosen an approach to Forth semantics which is prompted by the needs of our particular research, which is to lay foundations for formal verification of a compiler that uses Forth as its target language. Our source language, Ruth-R, is an expressive reversible language with a prospective values semantics. To allow the most direct comparison of meaning between a Ruth-R program and the corresponding RVM-Forth program produced by the Ruth-R compiler, we provide a semantics that is based on expressing the meaning of Forth operations and programming constructs in terms of PV semantics.

References

[1] J-R Abrial. *The B Book.* Cambridge University Press, 1996.

[2] E C R Hehner. *A Practical Theory of Programming.* Springer Verlag, 1993. Latest version available on-line.

[3] E R Hehner. Retrospective and Prospective for Unifying Theories of Programming. In S. E. Dunne and W Stoddart, editors, *UTP2006 The First International Symposium on Unifying Theories of Programming*, number 4010 in Lecture Notes in Computer Science, 2006.

[4] P. K. Knaggs and W. J. Stoddart. The Cell Type. In *University of Rochester Forth Conference on Automated Instruments, The Journal of Forth Application and Research (JFAR)*, June 1991.

[5] J F Power and D Sinclair. A Formal Model of Forth Control Words in the Pi-Calculus. *Journal of Universal Computer Science*, 10.9, 2004.

[6] W. J. Stoddart. An Event Calculus Model of the Forth Programming System. In *12th EuroForth Conference Proceedings*, October 1996. Note that the downloadable document contains the pages in reverse order.

[7] W. J. Stoddart and P. K. Knaggs. Type Inference in Stack Based Languages. *Formal Aspect of Computing*, 5(4):289–298, 1993.

[8] W J Stoddart and F Zeyda. A Unification of Probabilistic Choice within a Design-based Model of Reversible Computation. *Formal Aspect of Computing*, 2007. Published on line, DOI 10.1007/s00165-007-0048-1.

[9] W J Stoddart, F Zeyda, and S Dunne. Preference and non-deterministic choice. In *ICTAC'10. Proceedings of the 7th International colloquium conference on Theoretical aspects of computing*, volume 6255 of *LNCS*, 2010.

[10] W J Stoddart, F Zeyda, and A R Lynas. A Design-based model of reversible computation. In *UTP'06, First International Symposium on Unifying Theories of Programming*, volume 4010 of *LNCS*, June 2006.

[11] W J Stoddart, F Zeyda, and A R Lynas. A virtual machine for supporting reversible probabilistic guarded command languages. *ENTCS*, 253, 2010. Extended version of a paper presented at RC2009.

[12] F Zeyda. *Reversible Computations in B*. PhD thesis, University of Teesside, 2007.

Connection of a Forth target with a Forth host

Willi Stricker

Springe, Germany,

September, 12, 2012

Basic properties of host target communucation

As most modern micro processors/controllers the STRIP Forth processor [1] is equipped with a special boot program for programming and debugging. It is started by activating a special boot pin while restarting the processor (activating the reset pin).

This program is a very short but very versatile one, it can be used to connect common Forth targets to any (Forth) host in a very universal way. A Computer, for example a PC, controls a target system, for example an evaluation board.

In general the connection needs two programs, one inside the host and one inside the target (the latter is the boot program). Both of them are mostly identical but have some differences.

Demands for the cooperation:

The target should seem to be an integrated part of the host's own system.
The target contains program parts necessary for its own operation only.
All program parts that are used for configuration, management compilation programming etc. of the target instructions are inside the host.
The communication is done by calling target instructions (actual Forth words) by the host.

The complete operation is controlled by the host. The target is connected via an interface with the host (for example a serial interface like RS232 or USB, using the UART format and byte by byte transfer). The host is master, the target is slave. The host contains all auxiliary programs like compiler, interpreter, assembler, editor, disk system. It also contains the headers of the target's functions („name“ and „link“).

Execution of a target instruction by the host

1. The host sends the input parameters to the target,
2. the host sends the target instruction code to the target,
3. the target executes the instruction,
4. the target sends the manipulated output parameters to the host

Note for the parameters:
In a Forth System the parameters are on the parameter stack. The program has no information about the amount of parameters the instruction is needing for input and output. So the whole stack is sent to the target, it manipulates the parameters it needs, and sends the remaining stack back to the host. To minimize the data transfer the amount of parameters can be limited by the host (e.g. 10 or 16) to save time for the transfer.

Working course at execution of a target instruction by the host

host		Target
working		waiting for an instruction
sending parameters	=>	getting parameters
sending instruction	=>	getting instruction
waiting for response		executing instruction
receiving parameters	<=	sending parameters
continuing working		waiting for the next instruction

Course of the communucation

Byte order for the parameters: first low byte then high byte, If the count is zero, no parameters are transmitted.

1st byte	= start byte, always zero
2nd byte	= parameter count
3rd byte	= 1st parameter, low byte
4th byte	= 1st parameter, high byte
.	.
.	.
(2n+1)th byte	= nth parameter, low byte
(2n+2)th byte	= nth parameter, high byte

Transfer order:
The stack is always sent starting from down (bottom of stack BOS).
Conclusion: At sending the stack has to be rolled up from bottom. At receiving the parameters are simply pushed onto the stack.

Auxiliary Instructions and programs for the communication

Receiving and sending one byte from or to the interface hardware

These instructions are actually hardware dependant. It is just a possible example. It is assumed that there are two UART registers, a control register UACON and a data register UADATA. The control register has control bits for receiving selected by a receive mask RECMASK and for transmitting selected by a transmit mask TRAMASK.

Receive one byte:
```
: GETBYTE   ( -> byte )
    BEGIN
      UACON C@
      RECMASK AND
    UNTIL
    UADATA C@
;
```

Transmit one byte:
```
: PUTBYTE   ( byte -> )
    BEGIN
      UACON C@
      TRAMASK AND
    UNTIL
    UADATA C!
;
```

Handshake

A handshake is provided, it is mandatory because the time delay of the instruction execution of the target is unpredictible. Here it is done by software. a hardware handshake is basicly possible but not recommended because most interfaces don't provide a hardware handshake (e.g. evaluation kits).
The host (master) sends a byte and waites afterwords for another one of the target even if it expects one it has to send a dummy first. The target (slave) waites for a byte from the host and sends one afterwords even if it has to send one.

Host sends a byte with handshake:
```
: PUTCHAR ( char -> )
    PUTBYTE
    GETBYTE ( dummy = sent byte )
    DROP
;
```

Host receives a byte with handshake:
```
: GETCHAR ( -> char)
    0
    PUTBYTE ( dummy = 0 )
    GETBYTE
;
```

Target sends a byte with handshake:
```
: PUTCHAR ( char -> )
    GETBYTE ( dummy = 0 )
    DROP
    PUTBYTE
;
```

Target receives a byte with handshake:
```
: GETCHAR ( -> char )
    GETBYTE
    DUP
    PUTBYTE ( dummy = received byte )
;
```

Program for sending the parameters

```
: PUTPAR    ( n Parameter -> )
    0 PUTCHAR      send start byte
    SP@ DUP        parameter count
    PUTCHAR        send count
    BEGIN          loop for sending  the n Parameters
      DUP          count
    WHILE
      DUP
      PICK        pick next parameter
      DUP PUTCHAR send low byte
      CSWAP       swap bytes
      PUTCHAR     send high byte
      1-          decrement count
    REPEAT
    DROP
    0 SP!          stack initialisation
;
```

Program for receiving the parameters

```
: GETPAR ( -> n Parameters)
    BEGIN
      GETCHAR 0=     receive start byte + test
    UNTIL
    GETCHAR          receive parameter count
    BEGIN            loop for receiving n parameters
      DUP            count
    WHILE
      GETCHAR        receive low byte
      GETCHAR        receive high byte
      CSWAP OR SWAP concatenate to one word
      1-             decrement count
    REPEAT
    DROP
;
```

The instructions GETPAR and PUTPAR are identical for host and target, only the handshake instructions GETCHAR and PUTCHAR are different.

Target communucation program

The target uses a minimum Forth kernel, that at least contains the instruction for the communucation.
The program is an indefinite loop, reacting at demand by the host only.
The target obviously needs an initialisation of the Interface hardware, that sets the mode of the interface „PORTINIT“. This program is completely hardware dependant.

```
: COMMUNIC ( -> )
    PORTINIT ( -> ) port initialisation
    BEGIN           indefinite loop
      GETPAR        receive the parameters + instruction (cfa)
      EXECUTE       execute the instruction
      PUTPAR        return the resulting parameters
    AGAIN           return always
;
```

Note:This is the boot program of the STRIP Forth processor [1].

Construction of a target instruction in the host program

The target instruction in the host is made of a standard header (name and link) followed by an instruction (cfa of a Forth word) whose name is DOTARGET and followed by the code address of the target instruction (inside the target). Before target instructions can be accessed by the host, the interface has to be initialized.

| Haeder (target instruction name, link) |
| DOTARGET |
| target instruction address |

The host instruction „DOTARGET“ has the following definition:

```
: DOTARGET
    R>       get memory address from return stack
    @        get cfa of target instruction
    PUTPAR   send parameters to the target – instruction execution
    GETPAR   receive parameters from the target – result
;
```

Reference:

[1] Willi Stricker:
„A Processor as Hardware Version of the Forth Virtual Machine“; EuroForth 2011proceedings.

Building an LR parser for Pascal using Forth

Ian van Breda

Hailsham, East Sussex, UK

Abstract

The construction of a parser for Pascal is described. The parser is built by using Forth to include files that define the tokens and grammar of the language and may be thought of as a 'virtual machine' comprising a grammar code with lookup tables. The resulting parser can be run from within Forth or any other environment using simple drivers and is fully LR(1) capable. It is also well-suited for use with languages other than Pascal. An account is given of the author's early experiences with Forth at the University of St. Andrews and the Royal Greenwich Observatory.

1 Historical perspective

University of St. Andrews

I first came into contact with Forth when a colleague at the University Observatory in St. Andrews, Phillip Hill, came back from a visit to Kitt Peak National Observatory in Tucson, Arizona, excited about a newly invented computer language called Forth. A short time later, I used Forth on a spectrum scanner at the Cerro Tololo Observatory in Chile and on an infrared photometer at Kitt Peak, being very impressed with its ease of use and robustness.

At the time, much publicity had been given to the 'software problem', which alluded to the fact that, while there had been spectacular advances in hardware, there had been no equivalent in software.

Although Charles Moore had invented Forth originally for carpet manufacture, it was at Kitt Peak where it made its reputation on the 11-metre radio telescope that was responsible for the discovery of the great majority of molecules in interstellar space found in the early days of that subject. Using just a 16-bit minicomputer, it was possible to control the telescope, acquire observational data and perform data reductions for scientific analysis. This led to Forth's being dubbed 'the language of astronomy'.

We were fortunate to be able to purchase a system from Forth Inc. for a Data General Nova computer, the first Forth system outside the USA. This had the added attraction in St. Andrews that we were close to the Firth of Forth in Scotland: Charles Moore had originally intended to call it 'Fourth' for 'fourth generation language', but the original implementation could only take five-letter names.

This solved the 'software problem' for us instantly. We could have three students at a time working on data analysis using just one 16-bit minicomputer, helped greatly by the compactness of the Forth code. The very short learning curve for Forth meant that they were up and running very quickly in this new environment.

Around this time, there had been a move to use Camac, the interface standard used in nuclear physics, as a standard in astronomy: it was easy to access in software and had a number of existing cards for nuclear physics, of which one of the most interesting was a high-speed pulse counter for photomultipliers. This was connected to the Nova computer and allowed photon-counting photometry to be carried out in a remote dome, around 50 metres way, using a fibre-optic link.

The combination of Nova computer and Camac was also used in a scanning spectrometer. The Nova was later replaced by a PDP-11 computer that doubled as a built-in Camac crate controller.

Royal Greenwich Observatory

Subsequently I moved to the Royal Greenwich Observatory in Sussex (RGO), now sadly shut down after a spectacularly unsuccessful and ultimately fatal move to Cambridge.

Microprocessors were relatively new innovations at that point and we obtained a development system for the 8-bit Motorola MC6800 microprocessor to run a microFORTH system to control a high-speed photometer for use at an observing site in Spain.

Around the same time, a major ImageForth system was obtained from Forth Inc. to run on a PDP-11 computer attached to a scanning microdensitometer, which was run as a national facility for astronomers to scan photographs, both images of the sky and spectrograms. It was used to record scans on 7- and 9-track tapes for university users to take back with them for scientific analysis. It was also used for features like computer-assisted alignment of spectra.

The image processing system was ported to an MC6800 development system for work in the laboratory on an intensified photon counting detector, which was built for the South African Astronomical Observatory for use on their 1.9-metre telescope [1].

The image processing system was also ported to LSI-11 computers for use in the laboratory for development of liquid-nitrogen cooled solid-state detectors, particularly charge-coupled devices (CCDs), with a complete CCD camera being supplied to the Anglo-Australian Telescope. The controllers used to generate the waveforms for reading out CCD chips were based on bit-slice processors, also programmed in Forth. A summary of early developments in microprocessors using Forth at the RGO is given in [2].

Subsequently, detector development in the laboratory was switched to polyFORTH on MC68000 series processors running on VME bus. This was converted in-house from address threading to direct execution/subroutine threading on full 32-bit MC68020 and MC68030 processors [3]. For generating the wave-forms needed to read out CCD chips, MC68008 processors were also used. These were able to adopt the same software system with some minor conditional compilation to distinguish the two systems, mainly for arithmetic and the handling of CPU exception frames.

At the 4.2-metre William Herschel Telescope (WHT) on La Palma, VME systems running Forth were used for acquiring and processing CCD images, both for scientific astronomical detectors and the autoguiders at the various foci of the telescope. They were also used for the integrating intensified TV cameras needed to find and check that the desired faint object was being observed.

Individual fibre-optic links to specially designed VME cards, again programmed in Forth, were used to transfer detector data from the telescope, particularly to provide isolation against lightning strikes.

Ethernet, acting as a 'utility network' was used for the control systems on the telescope, as well as for individual instruments and sensors. Individual nodes on the network employed MC6809 processors programmed in Forth and were connected to the network by means of RS-232 links.

Image processing

The concept of the *ImageForth* image handling and processing system provided by Forth Inc. and installed by Charles Moore and Elizabeth Rather, is shown in Figure 1.

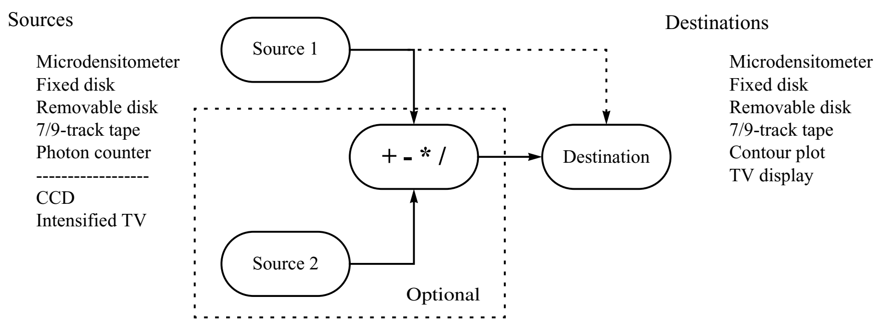


Figure 1. Schematic for the ImageForth image handling system.

In this very elegant scheme, various devices were able to act as image sources or destinations. Some could act only in one capacity (a contour plot on a storage tube could only be a destination, a CCD could only be a source). Others, such as a disk, could act in both roles. Sources simply had to be able to provide an image by reading one row at a time, destinations had to be able to write one row at a time.

An image 'pass' could be performed by transferring it directly to the destination. Optionally, it was possible to select two sources and perform arithmetic on them, such as dividing an image from a CCD by its 'flat-field' to allow for variations in sensitivity between the pixels.

Although not put forward as such, this represented an early application of object-oriented programming, with each device having methods for reading and writing rows.

Transferring to the Apple Mac

The MC680xx Forth was later ported to the Apple Mac, acquiring the name, Forth/68, where it proved to be possible to use pre-emptive multi-tasking, despite that fact that the native Mac OS has poor and complex task-switching and no in-built ability to run with command-line input. The parser described here has been developed on a PowerPC Mac Mini running in Classic mode. Although this is an emulation of the 680xx instruction set, it is easily fast enough to develop the parser, for which it takes less than one second to build all the necessary tables. The 680xx, being CISC, has the advantage of having an assembler that is very easy to use, unlike RISC processors. However, assembler is not needed for building a parser and the code used is ANS-compatible, although a front-end is required to make word lists compatible with Forth/68.

2 Early problems with Forth

Despite its considerable success, it was difficult to get Forth accepted by universities who used the facilities provided by the RGO. Some of the difficulties experienced with it were:

- Restricting names to three characters with counts led to some weird names with the inclusion non-alphanumeric characters. This led some to describe Forth as a 'write-only' language.
- Early systems crashed easily due to lack of syntax checking and complete system visibility.
- The use of Forth blocks for source code meant that the structure of definitions could be difficult to read.
- Initially there was no documentation embedded in the source code but later use of 'shadow blocks' that accompanied each source code block helped considerably.
- Stack operations could become very elaborate, again making reading of source code difficult.
- There was no support for type-ahead of the next command line during a long operation.
- Sometimes users were determined not to accept Forth, come what may.

The last point can be illustrated in the context of the William Herschel Telescope. This telescope was brought into operation immediately after primary commissioning, without the usual two-year bedding in period for telescopes of that size. This was made possible by lending one of our laboratory Forth systems to the La Palma observatory, as the telescope computers were well behind in their software development. It was on this Forth system that the telescope made its reputation by measuring the 'red shifts' (radial velocities) of infrared galaxies that had recently been identified by the IRAS infrared satellite.

However, in committee and without warning to us, one of the users of this facility described Forth as a 'pig'. It turned out that he had been given a list of definitions to use at the terminal that included hyphens, as is usual in Forth. However, he had tried to type these in as underscores and had not even bothered to ask before drawing his erroneous conclusion!

2.1 Advantages

Nevertheless there were many advantages:

- The user command-line interface is much better than either Unix or MS-DOS, which had given command-line input an unjustifiably bad name.
- While system crashes caused problems when some users were developing programs while others were trying to run fully operational programs, the simple solution was not to develop new programs while others were running operational programs.
- It was easy to develop programs for new instruments at the telescope, as these could be run by typing in primitive definitions then combining them into higher-level definitions later.
- Forth provides direct access to hardware, which makes testing much easier.
- Forth responds very rapidly to input commands.
- Pre-emptive multi-tasking made it possible both to run background tasks, for example to drive a cathode-ray display, while undertaking observations. It also meant that observations could be aborted in a controlled fashion when needed.
- Fully tested Forth applications are as robust as the very best programs written in other languages and are generally better.
- No bugs were ever reported on the WHT imaging systems on La Palma.

2.2 More recent developments

Since the early days of Forth there has been a number of improvements that answer most of the criticisms:

- There is plenty of memory space on 32-bit computers to allow full names to be used without having to resort to unusual characters.
- Use of word-processor text files for source code allows much improved documentation, including a brief description of the function of each definition, comment-per-line documentation and the use of indenting and phrasing within each definition.
- System crashes are greatly reduced through syntax checking and disappear almost completely if memory protection is also used.
- 'See-flow' debugging, which I believe was invented by Chris Stephens of Comsol Ltd, eliminates the vast majority of bugs, leaving only possible conceptual errors. In this scheme, source text is displayed alongside the current state of the stacks or registers, with a cursor marking execution progress. Execution advances by single steps by striking a key at the keyboard. The source code to debug is selected by enclosing it in a trace segment, while individual breakpoints can also be inserted. It is also possible to skip a specified number of breakpoints when there is a problem that only occurs after many definitions have been executed. This is much easier to use than the debuggers that come with the IDE development systems common with other languages. Surprisingly, it is not in as common use as might be expected.
- Type-ahead was added for the command line, which can also be edited, an essential for long names.
- LOCALS| greatly simplifies parameter stack operations, making definitions much easier to read.
- LOCAL definitions allow definition headers not needed in a running system to be discarded to avoid clogging up the dictionary with unused names (‘dictionary entropy’).

3 Why use other languages?

If Forth is so superior, why should other languages be used at all?

Firstly, although using a stack on a personal calculator is easier than the bracket notation common on other calculators, many users require a language that is more suited to the notation used in algebra, particularly when it comes to data structures. While algebraic notation can be used within the Forth context, it is usually circumscribed in its scope.

Another important factor is that there are many numerical algorithms written, particularly in C, that are very useful in image processing. For the RGO, there were many more users willing to program in C than in Forth, even though C sits uncomfortably as a high-level language with its terseness and quaintly eccentric choice of operators making it no easier to read than Forth, especially when it does not use comment-per-line documentation. Indeed its header files can be more difficult to read, its use of pointers is much less clear and often its logical expressions are more difficult to read than their Forth counterparts.

3.1 Why use Forth with these languages?

Forth provides a command-line interface that allows direct access to functions and procedures, making it both easier to use and to diagnose any bugs. It also adapts well to use with menus. Where other languages place excessive restrictions on data-typing, Forth can provide import routines to bypass such restrictions.

An important feature of Forth is that it can be used to generate the tables needed for parsing the strict grammars of other languages probably more easily than any of these languages themselves. This may seem surprising, given the free-wheeling nature of Forth. However, it is precisely the ability of Forth definitions to have names consisting of any group of characters that makes it possible uniquely to generate the tables needed for parsing by simply using `INCLUDE` to load the files that define the various aspects of the grammar and executing them directly.

An added bonus is that it is possible to make a parser fully LR(1) capable without the number of states getting out of hand. Normally, the number of these states is so large that special techniques are needed for reducing the number of states (and consequently also reducing the power of the parsing scheme).

3.2 Why Pascal?

The parser described here has been written to make it work easily with different languages, with the Forth code used to build the parser separate from the files that specify the syntax of the language.

Although C and its variants are used widely, Pascal was chosen because of its simplicity and because it wins hands-down over C on readability. This meant that problems of implementation could be solved within a simpler language set-up. C can, of course, be supported by implementing the necessary files that are used to define its tokens and syntax.

3.3 Problems with cross compatibility

A frequently heard criticism is that other languages cannot call Forth routines, although it is possible to call functions and procedures in other languages from Forth, with the appropriate stack adjustments.

This problem lies neither with Forth nor the other languages but in the implementation of those other languages using a single stack to house function/procedure arguments, local variables and the subroutine return address. Disadvantages of doing this include: building of the stack frame is more complex, and hence slower, than when using a separate return stack; the scheme does not match well to assembler subroutine calls, which usually transfer their arguments in registers but can also do so on a parameter stack.

The simple solution is to follow the Forth practice of using separate parameter and return stacks, making it possible for routines to be called both ways.

4 The function of a purser

The schematic workings of a compiler, of which the parser is a key part, is shown in Figure 2.

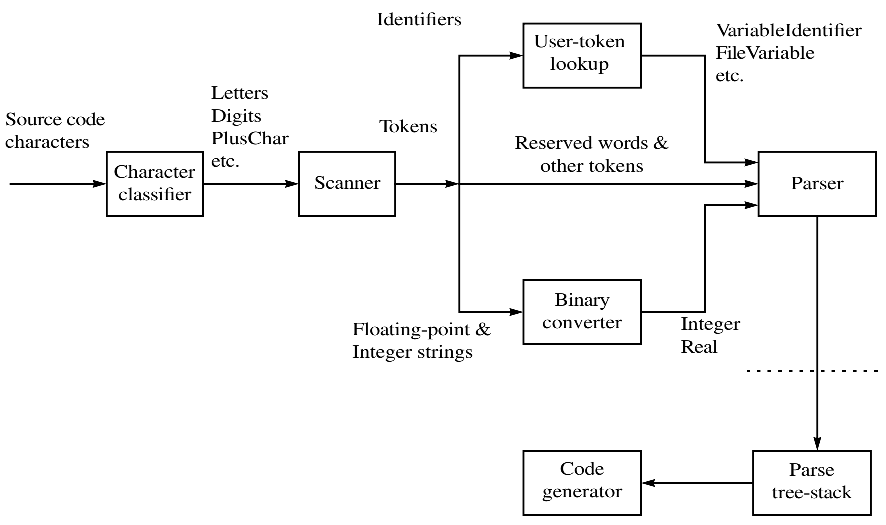


Figure 2. Outline operation of a compiler

Source code in the form of individual characters is passed to a classifier. This identifies generic letters and digits to simplify the process of scanning for tokens. It is implemented in the form of a simple lookup table.

The scanner extracts tokens from the input stream. This is different from the equivalent process in Forth where the tokens are simply obtained by looking for character strings that are separated by white space (spaces and tabs) and ends of lines. For example, it is possible to write a formula in Pascal without spaces between the tokens. Thus

```
x:=y+z;
```

contains six tokens which need to be identified separately.

Some tokens, such as reserved words, can be passed directly to the parser for checking that the grammar for the language is being followed correctly. However, other tokens, particularly identifiers and numbers, need to be grouped in some way, as a grammar cannot possibly be defined based on the actual contents of such tokens.

Instead, user-defined tokens are classified into named groups, i.e. token types. Thus, when defining a new field in a Pascal record, we need a new identifier (`NewIdentifier`) that names the field. However, when used later in the source code, the parser must recognise it as the name of a data item that has already been defined, `FieldIdentifier`. This process is handled by calls to the semantic routines that are used to 'decorate' the basic grammar and which join the parser to the real world of computing.

Tokens and token types are passed to the parser, which checks that they satisfy the grammar of the language. The parser also builds an intermediate representation (IR) of the program being compiled, with the help of the semantic routines. The IR can take a number of different forms. In Figure 2 it is shown as a 'tree-stack'. In such a scheme, *semantic records* are added to a stack as parsing proceeds and are then incorporated into a tree structure as each grammar production is recognised.

The tree can later be converted into code that can run on a computer either as assembler code or can even take the form of Forth.

The parser itself extends as far as the dotted line in Figure 2. This is a natural break point, since building of the parser requires considerable dictionary space both for Forth code and for the support tables needed. It is therefore better, in a running compiler, for the parser builder to generate the lookup tables needed by the compiler and save these to disk, where they can easily be picked up later.

This results in a considerable saving in the memory needed, even though memory on modern computers is very large by comparison: the parse builder uses around 125K bytes for 32-bit Forth but needs only 15K bytes for the parse tables when using 16-bit table-cell sizes.

So far the development has proceeded as far as the dotted line in Figure 2, i.e. the parser is complete for Pascal, with some elements of tree-stack building added for testing out on sample programs.

Using Forth

The tokens and grammar are defined using files specific to the language for which a parser is being created. These files are simply passed through the Forth interpreter, sometimes in multiple passes.

A key issue is that it is possible to define the symbols in these files as Forth words that can be executed by the interpreter. Apart from meta symbols which control the building process, the symbols normally carry a reference index: there is no need to decorate the files with explicit numerical values, often used in other environments.

It is important to bear in mind that there are three stages to consider here:

(i) Building the tables needed for parsing by interpreting the language-specific files 'on-the-fly';

(ii) Using lookup tables and grammar execution code when compiling a program;

(iii) Actually running a program after it has been compiled.

We are concerned mostly with (i) here and how it relates to (ii).

The parser tables are built by including the Forth source code files interspersed with the files that define the syntax of the target language; the parser builder does not exist as a separate program.

5 Scanner

The scanner consists of two parts, the character classifier and token extraction. The notation used here follows closely that given by Fischer & LeBlanc in the original edition of *Crafting a Compiler* [4].

5.1 Classifying characters

In languages like Pascal, we need to classify source-code characters for building tokens. A partial listing of the file used to build the lookup table for performing this classification is shown in Listing 1; the numerical codes are ASCII-specific. This follows loosely the notation given by Fischer & LeBlanc for input to the

ScanGen scanner generator program.

The difference here is that all the words are Forth words that either define new names or help to build the lookup table when executed. The words must also be separated by white space or end-of-lines.

```
CharacterClasses
   CharClass Tab         =  09
   CharClass EndLine     =  10 , 13
   CharClass Blank       =  32
   CharClass LetterE     =  E , e
   CharClass OtherLetter =  A .. D , F .. Z , a .. d , f .. z
   CharClass Digit       =  0 .. 9
   CharClass Star        =  *
   CharClass PlusChar    =  +
   ..........................
   CharClass UpArrow     =  ^
   CharClass SingleQuote =  '
EndCharacterClasses
```

Listing 1. Part of the file used to classify characters in Pascal

Before the Pascal-specific file can be included, we need definitions for the Forth words that define the named categories and also those that build the character-class lookup table. For this purpose, two vocabularies (i.e. named word lists [5]) are used, one for the words that do the classifying (`CLASSIFY`), one for the vocabulary that contains the class definitions (`CHARACTER-CLASSES`). Each such definition has a numerical-index in its parameter field, assigned in chronological order as it appears in the file.

The Forth source code for achieving this is shown in the partial listing given in Listing 2. Ada line-comments, prefixed by `--` are used for explanatory comments; they are also used for titles where related definitions are grouped together.

`CharacterClasses` is defined in the `FORTH` vocabulary, but the words that do the actual classifying are defined in the `CLASSIFY` vocabulary, The Pascal-specific file, as given in Listing 1, is then loaded by including the file whose name string is `$CharacterClasses`.

On subsequent execution of `CharacterClasses`, when the Pascal file is included, the compilation vocabulary is set to `CHARACTER-CLASSES`. The search order is set to the `CLASSIFY` vocabulary, searched first, followed by `FORTH-HOOKS` which is a vocabulary containing some very basic Forth words, including \ to allow comments to be included in the text, and a few 'get-out-of-jail' words, such as `EMPTY`, which allow basic recovery if things go wrong during loading.

`CharClass` defines a named classification using `CREATE`, with a running chronological index (`CharacterClassCount)` in the parameter field of the defined word; the vector code is not executed at this stage.

When building the classification lookup table, whose address is set in `CharacterClassTable`, the equal and comma definitions use `GET-CODE` to find the code for the next character in the input stream if a single character. If the input item is a double character, the numerical code is returned (ASCII here). This code is used as an index into the table by `SET-CLASS` to insert the current class index into `CharacterClassTable`.

The double-dot not only sets the class code in the table for the next character/number-pair but also fills in the gaps between it and the previous table entry. Thus `A .. D` fills in all of the table elements for A, B, C, D with the current character class.

Several classifications, such as `Plus`, contain only a single character.

5.2 Building tokens

Before considering how the Pascal tokens are extracted from the input stream, a file is included giving names to all the Pascal tokens and token types. A partial listing is given in Listing 3, as the file is too big to include in its entirety here. Once again the file is executed in the normal way, as for any Forth source-code file, by using `INCLUDE` to load it.

```
   \ Initialising character class look-up table
: CharacterClasses   CharacterCount       \ Element count/size-in-bytes
   CHAR_CELL                              \ Size of each element in bytes
   0                                      \ No auxiliary parameters
   1 NEW-TABLE  DUP TO CharacterClassTable
                                          \ New dictionary-aligned 1-D table
   ERASE-TABLE                            \ Fill with default zeros
   CHARACTER-CLASSES DEFINITIONS          \ Compile to CHARACTER-CLASSES
   VIA  FORTH-HOOKS CLASSIFY  ENDVIA ;    \ Search CLASSIFY/emergency exits


CLASSIFY DEFINITIONS  VIA  FORTH  ENDVIA
   \ Place definitions below in CLASSIFY but search FORTH on its own.
   \  This means that the CLASSIFY definitions below cannot call each other

   \ Making a new class category
      \ Expects: previous class number; class name in input stream
      \ Returns: new class number
: CharClass ( n $ -- n|)
   CharacterClassCount                    \ Count is index code for class
   DUP CREATE,                            \ Define character class as a
      \  constant offset in scan actions table: 0=illegal
   DUP TO CurrentCharacterClass           \ Save as class being processed
   1+ TO CharacterClassCount              \ Advance current class total
   DOES> ( -- addr)                       \ Tail code expects parameter addr
      CharacterClassVector EXECUTE ;      \ Execute table-building code

   \ Starting specification of a class
      \ Expects: single ASCII character or numerical code in input stream
: = ( $ -- char/u)
   GET-CODE                               \ Next character/numerical code
   SET-CLASS ;                            \ Enter class number into table

   \ Continuing class specification
      \ Expects: character/numerical code in input stream
: , ( char $ -- char/u)                   \ Enter class for next item
      -- This = is the CLASSIFY version
   GET-CODE                               \ Next character/numerical code
   SET-CLASS ;                            \ Enter class number into table

   \ Making multiple insertions in classification table for a range
      \ Expects: ASCII character/numerical code in input, last item in range
: .. ( $ -- char2/u2)   CharacterCode 1+  \ Bump first to avoid double entry
   GET-CODE                               \ ASCII/numerical code, last char
   1+ SWAP  2DUP > NOT  ABORT" Characters in wrong order"
                                          \ Abort if not in ascending order
   DO  I SET-CLASS  LOOP ;                \ Fill in table entries for range
      -- This omits first item in range, else SET-CLASS would abort

   \ Terminating character class defining
: EndCharacterClasses
   FORTH DEFINITIONS ;                    \ Restore basic compile/search

FORTH DEFINITIONS                         \ Restore basic compile/search

$CharacterClasses INCLUDED                \ Build character-class defns.
```

Listing 2. Building the character-class lookup table

```
-- Single-character tokens --
     Token '+'         Token '-'         Token '*'         Token '/'

-- Double-character tokens --
     Token '<='        Token '>='        Token '<>'        Token ':='

-- Reserved words --
     Reserved and          Reserved array          Reserved begin

-- User-defined token types --
     UserToken DigitSequence            UserToken UnsignedReal
     UserToken Identifier               UserToken NewIdentifier
```

Listing 3. Partial listing of the Token List file for Pascal

`Token` is a predefined word that gives a name to a token for use when interpreting the formal grammar for Pascal; these definitions are placed in a `TOKEN-LIST` vocabulary. The names are enclosed in quotes to differentiate them from the meta characters used in the formal definition of the grammar.

`Reserved` places definitions for the reserved words in their own vocabulary, `RESERVED-WORDS`. The reason for this is that, when parsing, the reserved words need to be searched first before any other definitions, by placing `RESERVED-WORDS` at the top of the search order.

`UserToken` defines *token types* for tokens defined by the user and are also held in `TOKEN-LIST`. These are identifiers and numbers which need to be categorised by kind for processing in the grammar.

As with character classes, the defined words are given an ordered index and use vectored code for execution later when processing the Pascal grammar file.

It will be seen that, for example, the plus character appears in three guises: as `+` in the source code of a program; as `Plus` for building tokens; and as `'+'` in the formal grammar. This might seem overkill but it does make the various files easier to read and, in the formal grammar, is essential to differentiate tokens form the meta symbols.

In the literature, token building generally uses so-called *regular expressions* to build what is somewhat grandiloquently called a *finite automaton*. In such an implementation, each token/token-type is defined by a regular expression which is then used to build a lookup table. The automaton simply goes from state to state according to the class of each character in the input stream; there is no nesting.

The disadvantage of using regular expressions is that some token definitions overlap, such as integers and floating-point numbers, so these need to be spliced together creating some complication for the code used to build the lookup table.

Instead, we use a scheme based on the common EBNF (Extended Backus-Naur Form) developed originally for processing Algol grammars. In this, each token-building definer must start with a character or characters that are unique to it.

The extended EBNF meta characters have the following meaning:

```
|         -- alternatives
( ... )  -- grouping
[ ... ]  -- single option
{ ... }  -- multiple option
.         -- continuation
```

Extra definitions are added for processing the tokens

```
{Toss}   -- discard character
{Scrub}  -- discard complete token
#Token   -- specifies a token type
```

A partial listing of the file used to define how tokens are built is shown in Listing 4.

As before, the required lookup table is built with the help of Forth words, defined specifically for the purpose, by including the file. For example, opening brackets of various types start a new line in the lookup table; some extra processing is needed, however, to removed duplicate and redundant rows.

All `Token:` and `Gluon:` definitions start a new row in the scanner lookup table. `#Token` expects the name of a token or token type following in the input stream and inserts a marker for that token/type.

This generates a lookup table for which a partial listing is shown in Listing 5.

```
-- Macros --
     : Letter   ( LetterE | OtherLetter ) ;

-- Anonymous white space --
     Token:   { WhiteSpace } {Toss}

-- Simple non-alpha tokens --
     Gluon:   PlusChar '+'
     Gluon:   Less '<'  .  [ Greater '<>' | Equal '<=' ]

-- Compound tokens --
     Token:   Letter #Token Identifier .
     { Letter | Digit } #Token Identifier
```

Listing 4. Partial listing of file used to define how tokens are built

```
 Action codes
    REUSE-CHAR = 0                         APPEND-CHAR = 1
    NO-APPEND = 2                          SCRUB-TOKEN = 3
    Class         Do  Token         Next   Class         Do  Token         Next
 Row 0
    Default       2   ErrorToken       0   Unprintable   2   ErrorToken       0
    EndFile       2   EndOfFile        0   Tab           2   xx               0
    EndLine       2   xx               0   Blank         2   xx               0
    LetterE       1   Identifier       5   OtherLetter   1   Identifier       5
    Digit         1   DigitSequence    8   Star          1   '*'              0
    PlusChar      1   '+'              0   MinusChar     1   '-'              0
    Slash         1   '/'              0   Equal         1   '='              0
    Less          1   '<'              1   Greater       1   '>'              2
    DotChar       1   '.'              3   CommaChar     1   ','              0
    ColonChar     1   ':'              4   Semi          1   ';'              0
    LParenChar    1   '('             13   RParenChar    1   ')'              0
    RBracket      1   ']'              0   LBracket      1   '['              0
    LBrace        2   xx              14   RBrace        2   ErrorToken       0
    UpArrow       1   '^'              0   SingleQuote   2   xx               6
 Row 1
    Equal         1   '<='             0   Greater       1   '<>'             0
 Row 2
    Equal         1   '>='             0

 Row 5
    LetterE       1   Identifier       5   OtherLetter   1   Identifier       5
    Digit         1   Identifier       5
```

Listing 5. Partial listing for the lookup table used to build tokens and token types

The table is two-dimensional with rows corresponding to states of the automaton and columns corresponding to the input character-code classification. Each cell in the table contains three fields: what to do with the character, token index code if any; and the next row to go to. Only non-default table cells are shown.

As each character is taken from the input stream and classified, the character itself is normally added to a string which will ultimately form the required token string.

The action to take is looked up in the table row that corresponds to the current state of the automaton. It may be to re-use the character without adding it to the token string, extract the string from the input stream and add it to the token string, extract it from the input but discard it (as in character strings delimited by quotes that don't form part of the string as such) and erase the current token (as happens with comments which, in this implementation, are not passed to the parser).

The default table cell is to re-use the character without extracting it from the input stream, with no token set and destination row zero. Moving to row zero also indicates that no further characters may be added to a token.

Processing of a token always starts at row zero with an empty token string. Thus, if an unprintable character is encountered, it is not added to the token string, but generates a special `ErrorToken` and returns to row zero indicating that the token is complete. In this case the token string is empty.

If a `<` symbol, which has the classification `Less`, is encountered at the start of a token, it is added to the token string, and control moves to row 1. If followed by a character classified as `Equal` or `Greater`, the character is added to the token string and control is transferred to row zero, indicating that the token is complete. Any other character causes a return to row zero, without adding to the token string.

We thus end up with possible strings `<`, `<=` or `<>` for the tokens themselves with the scanner returning index values for the named tokens `'<'`, `'<='` or `'<>'`, as applicable.

When a character `LetterE` or `OtherLetter` is encountered, the token is set as being an `Identifier` and control is transferred to row 5. If a letter or digit comes next, the character is added to the token string and the token again noted as an `Identifier`, with the token data updated. Processing continues at row 5. For any other character class, control is returned to row zero, indicating that the token is complete. In this case, both the token-type, `Identifier`, and the token string itself are returned, as the former is needed for checking the grammar while the latter is used to identify the actual identifier name.

The above scheme means that the longest possible token is always taken. Possible ambiguity is avoided by not always marking a string as a possible token. If we take the example of the string 100..200, this represents a range in Pascal. At the point where 100 is input, the token can be a `DigitSequence` (integer). However, when the dot following is found, the string could be part of a floating-point (real) number, 100.9, say, so the dot is added to the token string, which now has four characters, but the separate token code, character count (three) and saved `>IN` pointer are not updated.

For a real number, the dot must be followed by a `Digit` character. However, in this case it is followed by another dot, which is not allowed in a floating-point number. Control is therefore returned to row zero, indicating that the token is complete. The token-string count is backed off to three, `>IN is` backed off to before the first dot and a `DigitSequence` is returned with a token string containing just the 100.

6 Parsing

The function of a parser is to make sure that the source code for a program satisfies the syntax of the language. It also has the task of building an intermediate representation for the program.

6.1 Grammar notation

Grammars are often illustrated in the form of syntax diagrams, which can be very helpful in understanding the way in which the grammar works. However, this is not too much help in writing a parser for which we need a formal textual method of describing the grammar. This is done with a list of *productions* expressed in EBNF notation. The start of the list for Pascal is shown in Listing 6.

```
Productions

<start>                        -> <program> ( EndOfFile | endPascal ) ;

<program>                      -> <program_heading> ';' <block> '.' ;

<program_heading>              -> program <new_identifier> #ProcedureId
                                  [ <program_parameter_list> ] ;
```

Listing 6. Start of productions for Pascal

The left-hand side of each production takes the form of a named *nonterminal*, which specifies some construct in the language. The nonterminal is said to *produce* the right-hand side, which consists of a mixture of other nonterminals, *terminals*, which are the tokens and token types for the language, and meta characters which specify the way in which the production works in the context of the language.

The grammar is described as context-free, since, whenever a nonterminal occurs on the right-hand side of a production, it always has the same meaning.

The notation used follows closely that used by Fischer & Leblanc [4], adapted so that the file can be included in Forth, with all words delimited by white space and end-of-lines. Nonterminals are included in hairpin brackets for visibility in plain-text files. In order to enable nonterminals to be defined in the Forth dictionary, compound names use underscores as separators.

For the terminals, reserved words appear as they are, while special symbols appear in quotes and user tokens are identified by their token types. Semantic-routine references appear by name with a hash symbol in front. The hash has no special significance, except that it catches the eye in a plain-text listing.

The EBNF meta symbols have similar meanings to those used for defining how tokens are constructed (Section 5.2) but there is no dot for continuation. The symbol `->` starts the body of a production and a semicolon indicates its end.

Items in parentheses invariably represent a list of options separated by alternation symbols (vertical bars). Items in square brackets represent a single option, which may or may not be present in the source code being parsed. Thus a program parameter list may not be present in a program header (Listing 6).

Multiple options are included in braces. In the production

```
<label_declaration_part>      -> [ label DigitSequence
                                 { ',' DigitSequence } ';' ] ;
```

there may be nothing in the label declaration part (because the entire production is enclosed in square brackets). However, when present, it must start with the reserved word, `label`, followed by a `DigitSequence`. This, in turn, may be followed by as many `DigitSeqence` terminals representing other labels as we like, each preceded by a comma. A semicolon in the source code terminates the list.

6.2 The parsing process

Parsing of a program proceeds by looking up the next token/type in the input stream and using it as a terminal *lookahead*, starting at the `<start>` production, Listing 6. A Pascal program must start with the reserved word `program`. As implemented here, this is accepted as a valid lookahead by the first production, which nests a return point past `<program>` and goes on to the start of the second production. Again `program` is a valid lookahead, so control is passed to the start of the third production.

This can be represented by a series of *configurations*:

```
<start>                       -> <program> ● ( EndOfFile | endPascal ) ;

<program>                     -> <program_heading> ● ';' <block> '.' ;

<program_heading>             -> ● program <new_identifier> #ProcedureId
                                 [ <program_parameter_list> ] ;
```

Listing 7. Initial configurations in the Pascal grammar

where the various points are marked with dots (blobs), with the first two being nested on a parse return stack.

At this point we encounter the `program` token at the start of the production. This is checked against the current lookahead and control is passed to the point after `program`. A new terminal is extracted from the input stream which, in this case, must be a `NewIdentifier`:

```
<program_heading>             -> program ● <new_identifier>
                                 [ <program_parameter_list> ] ;
```

The production for `<new-identifier>` is now processed and parsing proceeds as before.

After processing any arguments in the program parameter list, if present, the end-of-production unnests the parse return stack and control is transferred back to the `<program>` production, as in Listing 7. Here, a semicolon is expected in the input stream and a check is made against the lookahead terminal; if they don't match, a syntax error is declared. If they do, the parser goes on to process `<block>` which does most of the work of parsing the program. It must be followed by a full-stop (period).

Once the `<program>` nonterminal completes, the parse return stack is unnested and control is returned back to the `<start>` production, where the next terminal must either be an end-of-file or `endPascal`, which is an extra reserved word added for returning to Forth processing.

At the end of the parse, some form of intermediate representation is returned. Here it is envisaged that it will be an *abstract syntax tree*, which is probably the simplest form to build and is also good for optimisation. The ultimate goal, though, is to produce assembler code for a target processor.

6.3 LL vs. LR parsing

The Forth programmer will immediately see a close resemblance between grammar productions and colon definitions in the way that they work when executed.

However, there are a couple of important reasons why we can't simply convert a grammar production to some form of colon-style definition. Firstly, a given nonterminal may have more than one production. For example

```
<adding_operator> -> '+' | '-' | or ;
```

is the equivalent of three productions, which can also be written as

```
<adding_operator> -> '+' ;
                  -> '-' ;
                  -> or ;
```

Listing 8. Splitting up alternative productions

Thus `<adding_operator>` can accept any one of three operands as its lookahead token and a selection mechanism is needed for determining which production to follow. Here it is done by means of a *parser lookup table*. The table has rows corresponding to nonterminal index, columns corresponding to terminal index and entries being the starts of productions in the grammar.

So far we have described a parsing technique known as top-down, predictive or recursive descent parsing. It is also known as LL(1) parsing, as the parse processes the source code input from left to right and the derivation also proceeds from left to right in the grammar, with a single terminal lookahead. In principal, it is possible to use more than one terminal as a lookahead: LL(*k*) denotes recursive descent parsing with *k* lookahead terminals but the lookup tables become impractically large. In what follows we shall take LL to mean specifically LL(1).

However, there are cases where choosing which production to select is ambiguous. Thus in a function declaration

```
<function_declaration> -> <function_heading> ';' <block> |
                          <function_heading> ';' <directive> |
                          <function_identification> ';' <block> ;
```

Listing 9. A nonterminal with a prediction conflict between productions

there is no way of predicting which of the of the productions applies in a particular situation, as all three accept the reserved word, `function`, as the lookahead terminal: `function` is the only terminal in the *first set* of each production, i.e. the set of terminals that can be accepted at the start of a production.

The solution adopted in LR(1) parsing (Leftmost parse, Rightmost derivation, single lookahead terminal) is to process productions which conflict in this way in parallel; parsing proceeds until a complete production is recognised. This is also known as bottom-up parsing or shift-reduce parsing from the way that it processes the parse stack often used in other implementations. As with LL parsing, in what follows we shall take LR to mean pecifically LR(1) parsing.

By way of illustration, we consider just the first two productions, where the conflict is most obvious. If these are processed in parallel then we have an LR state represented by

```
<function_declaration> ->● <function_heading> ';' <block> ;
                       ->● <function_heading> ';' <directive> ;
```

LR configurations also include the follow set, i.e. the full set of terminals that can follow the left-hand-side nonterminal in the grammar. This is because the productions may need to be processed in parallel until one or more productions reaches its end-of-production, in which case the follow set may be needed to resolve any conflict. If that is not possible, we have a so-called *shift-reduce* or *reduce-reduce* conflict and the grammar must be modified to avoid this.

In the case above, we do not need to go so far, since, when we eventually get past the semicolon token in the two productions,

```
<function_declaration> -> <function_heading> ';' ● <block> ;
                       -> <function_heading> ';' ● <directive> ;
```

the parser will accept, as lookahead tokens, any of `begin`, `const`, `function`, `label`, `procedure`, `type` and `var` for `<block>` and `Identifier` for `<directive>`. Since the two sets do not overlap, the parser can determine uniquely which production to select.

At this point, the LR state has split into two separate productions, which might be described as *singleton states* and which can be parsed using the simpler LL method.

6.4 Encoding the grammar

Named nonterminals

Before approaching how to represent the grammar, we first need to assign numerical values to the various named nonterminals that appear in the grammar productions.

This is done by making a pass on the grammar productions file using the techniques shown in Listing 2. In this case the meta characters and symbols have empty definitions, while the vector codes for the terminals are also set to do nothing. Any word that cannot be recognised is checked to see that its name is enclosed in hairpin brackets. If so, it is added to the dictionary and given a running index, which is later sorted alphabetically to help with displaying the grammar from within the Forth parser builder program.

Brackets

It is common practice to rewrite productions containing bracket pairings to avoid their use. For example,

```
 <statement_sequence> -> <statement> { ';' <statement> } ;
```

can be rewritten by adding a new production as

```
 <statement_sequence> -> <statement_sequence> ';' <statement> ;
                      -> <statement> ;
```

However, there are two significant disadvantages to this: firstly, it requires a major rewrite of the Pascal grammar to convert it to this form, as there is a large number of brackets of various types in the Pascal EBNF grammar; secondly, as seen above, the nonterminal is changed from being able to be processed using LL to the more complex LR processing.

The way chosen here is to treat each opening bracket as an anonymous nonterminal in its own right that 'owns' the production(s) enclosed within the bracket pairing. These nonterminals are given ordered index values that follow immediately after those for the named nonterminals (in hairpins).

Grammar coding

We can then construct a version of the grammar that can be executed when parsing a program, rather in the manner of a series of interpretive Forth colon definitions for the individual productions.

In this representation, the productions are laid out end-to-end as a series of *grammar items*, in the order that they are given in the grammar, but without the left-hand nonterminals. Every symbol in the right hand side of each production is represented by a grammar item, consisting of two or more 16-bit cells (they can be compiled as 32-bit, if desired).

The format of these items is shown in Figure 3. Each item starts with an item 'class', which specifies its type. This is followed by one or more parameters. For a terminal it is the token or token-type index, generated when the token list was loaded, as in Listing 3. For (named) nonterminals it is the index generated in the first pass on the grammar. For a semantic item, it is an index into a list of semantic routines that are to be executed as parsing proceeds.

The class specifies the type of action to take in each case. In a sense, we have a 'virtual machine' with each class consisting of a machine instruction, followed by a series of parameters that define the detailed behaviour of each instruction through the medium of lookup tables.

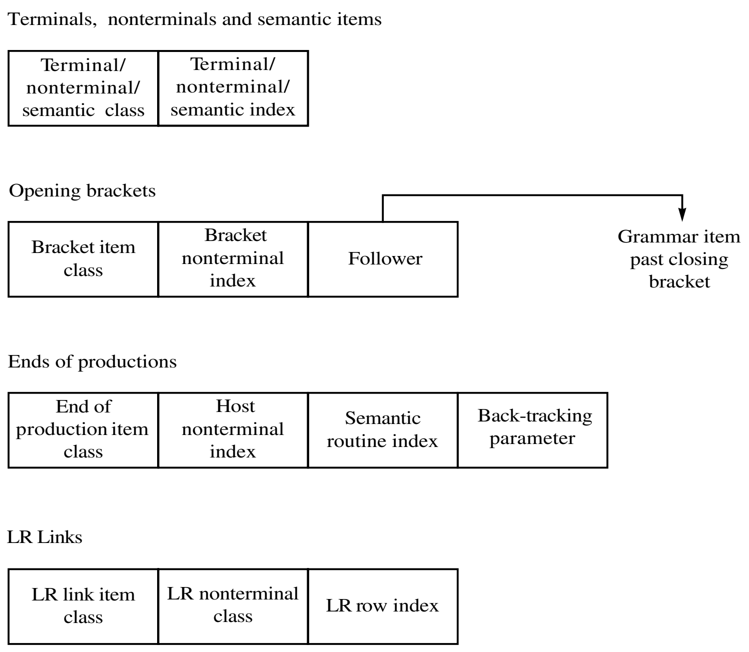

Terminals, nonterminals and semantic items
Terminal/ nonterminal/ semantic class
Terminal/ nonterminal/ semantic index
Opening brackets
Bracket item class
Bracket nonterminal index
Follower
Grammar item past closing bracket
Ends of productions
End of production item class
Host nonterminal index
Semantic routine index
Back-tracking parameter
LR Links
LR link item class
LR nonterminal class
LR row index

Figure 3. Structure of items in the grammar code

For an opening bracket, the class is specific to the type of bracket: parenthesis, square bracket or brace. The index following is that for the anonymous nonterminal for that bracket, while the follower gives the offset past the matching closing bracket within the grammar coding.

End-of-production items are generated for terminating semicolons, closing brackets and alternation symbols. The index for a semicolon is that for the host named-nonterminal on the left-hand side of the production. For a closing bracket, it is the index for the opening bracket, treated as a nonterminal. For an alternation symbol, it is the either the index for the left-hand named nonterminal, if not inside a bracket pair, or that for the opening bracket, if inside a bracket pairing.

The third parameter is the index for an optional semantic routine when it occurs at the end of a production.

It is assumed that, when parsing, each nonterminal will add a semantic record to the parse tree-stack. However, it is not always possible to guarantee that such a record will be present when an optional nonterminal construct that derives the empty string is not present. By backtracking, it is possible to enter an empty semantic record on the parse tree-stack of the right type, so that any semantic routine executed at the end of a production can always expect a given number of nonterminals on the stack for that production.

LR processing is slightly different in that two classes are used, followed by an index for the LR state.

Sets

In order to be able to build the required parse lookup tables, an additional pass is first made on the grammar productions text file to build a grammar table coded as given in Figure 3 but ignoring semantic references and assuming that none of the nonterminals needs LR processing. We call this the ‘LL grammar’ and building these tables is then done through one or more passes on this variant of the grammar in memory, rather than the text file. In all cases the pass is repeated until there are no more changes found.

This allows some important tables to be defined. Firstly, we need to know which nonterminals ‘derive the empty string’, i.e. are optional and may be absent in the source code for a program. An example of this is:

```
<empty_statement>              -> ;
```

which has an empty production. Or

```
<constant_definition_part>     -> [ const <constant_definition> ';'
                                   { <constant_definition> ';' } ] ;
```

where the construct may or may not be present in a program. On the first pass on the LL grammar code, both of these nonterminals will be marked as deriving the empty string.

Because `<empty_statement>` is in the list of productions for `<simple_statement>`, which can be written:

```
<simple_statement>     -> <empty_statement> ;
```

`<simple_statement>` will only be marked as deriving the empty string on the second pass, since the status of `<empty_statement>` is not known first time round.

Two other sets are essential in building the parse tables. The first set is the set of lookahead terminals that can start a nonterminal. These sets are held as bit patterns in an array indexed by nonterminal parameter value. For example, in Listing 6, it can be seen immediately that the reserved word, `program`, is in the first set for `<program_heading>` and will be added to the list of first sets on the first pass. However, it will only become apparent that `program` is also in the first set for the `<program>` nonterminal on the second pass. Again, passes are repeated until there are no further changes to the first sets.

Once the derives-the-empty-string and first-sets tables have been set up, it is possible to find the follow sets for all the nonterminals, again by multiple passes on the LL grammar code.

All of this then allows those nonterminals to be found that need to be treated by LR parsing because there are conflicts between the first sets (or first and follow sets if the nonterminal can derive the empty string).

Bracket nonterminals may also need to be treated using LR methods. For example, in

```
<case_statement>       -> case <case_index> of <case> { ';'<case> } [ ';' ]
                          end ;
```

the brace nonterminal must be treated using LR, as the production itself starts with `';'` which is also in its follow set, so when `';'` is encountered as the lookahead, the parser will not know beforehand whether this is the beginning of another `<case>` construct or the optional `';'` at the end.

6.5 Building the final grammar execution code

All of this now allows the final grammar code table to be built. This time, semantic routines are included, along with LR links for those nonterminals and brackets needing LR processing. The index field for each of these is set to the index values for an LR state., which are given index values following on from those for named nonterminals and bracket nonterminals. The LR link replaces the normal nonterminal/bracket grammar item.

There is one LR state for each LR link item in the grammar. Each LR state must then be ‘closed’ by looking at all productions and other LR states that the state can link to, with new LR states being added, as needed. Each of these is given its own index and an LR link that is added to the end of the grammar execution code in the form shown in Figure 3.

In the Pascal grammar, as written in Jensen & Wirth [6], there are only 11 LR links within the main body of the grammar rising to 25 LR states altogether when these are closed.

Two parse tables are built, one has rows that correspond to index values for named nonterminals, bracket nonterminals and LR state index values which are tagged onto the end of the terminals, one per LR state. The columns are numbered by index for the lookahead terminal. Each entry contains a position in the grammar code.

The second table is only used in the LR parsing process to look up the return point once a production has been recognised in the parsing process. It has rows corresponding the LR state index and columns matching the nonterminal index for the production in question.

As these tables are very sparsely populated, they are implemented in compact form using double-offset indexing as described in [4].

6.6 Grammar item actions

Associated with each grammar item is a 'driver' specific to that item. The different forms for the brackets give rise to three different grammar-item classes, making a total of nine altogether. These drivers are language-independent and so only need to be written once. They also have much code in common.

This gives a very simple scheme for executing a parse. We begin with the `<start>` production and a zero on the parse return stack. For each grammar item, the drivers carry out the following actions:

`Terminal`
When a terminal grammar item is processed, the driver looks to see if there is a match with the current lookahead terminal. If so, control is transferred past the item, otherwise a syntax error is declared.

`Named nonterminal`
For a named nonterminal, control is transferred to the start of the production that matches with the terminal lookahead, by looking it up in the main parse table. The return position past the grammar item is saved on the parse return stack.

If the nonterminal derives the empty string (i.e. is optional in program source code) and the terminal is not in the start set of the nonterminal, control is transferred past the grammar item without any nesting. If the terminal is not in the follow set, a syntax error is declared.

`Opening parenthesis`
This is very similar to that for a named nonterminal, except that, if the bracket-nonterminal can derive the empty string and the terminal is not in the start set but instead is in the follow set, control is transferred past the closing parenthesis.

`Opening (square) bracket`
Again behaviour is very similar to a parenthesis. If the lookahead terminal is in the first set for the bracket nonterminal, control is transferred to the start of the matching production (normally just past the opening bracket) and the location past the closing bracket is nested on the parse stack. Thus control will be passed to that point, once processing of the production has been completed.

Since bracket pairs always indicate a single-option for a program, if the lookahead terminal is not included in the start set of the production but is in the follow set for the bracket nonterminal, control is transferred past the closing bracket without any nesting. Again, a syntax error is declared if the lookahead terminal is in neither the start set nor the follow set for the bracket nonterminal.

`Opening brace`
This is almost identical to the behaviour for an opening bracket. The difference is that, when a match is found with the start set for the brace nonterminal, it is the point before the opening brace that is saved on the parse return stack, so the process will be repeated until a terminal is found that is not in the start set.

`Semantic reference`
The driver for this type of item uses the index in the grammar item to look up a semantic routine whose address or Forth token can be found in a table of references to the semantic routines for the language. This is the case where the semantic call is located within the body of a production, which must be in LL mode at that point for it to work.

`End of production`
If backtracking is needed because there is a nonterminal in the production that can derive the empty string or is an option bracket, the driver works backwards through the parse tree stack and inserts an empty semantic record into the parse tree-stack if the nonterminal is not present. It then executes the semantic routine, if there is one, which can then expect a fixed number of items on the parse tree-stack.

The semantic routine must leave a single nonterminal on the tree-stack. If required, it may need to add a semantic record, e.g. when an operator is encountered, as in Listing 8, there must be a semantic routine, e.g. `#AddOp` (not shown in the listing), that adds a semantic record for the operator using the most recent token extracted from the input stream to specify its type.

At the end, the driver marks the top item on the tree-stack with the index for the enclosing nonterminal, LHS nonterminal or bracket nonterminal, as appropriate.

`LR Link`

Processing of LR links is only slightly more complex than their LL counterparts and two drivers are used basically for looking up two different parser tables.

In the first case, the main parse table is looked up by row (LR state index) and column (terminal lookahead) and control is transferred to the point in the grammar code specified in the table, which can be to another LR-state link, or other grammar item, in which case processing will continue in LL mode.

However, the return destination saved on the parse return stack corresponds to the second class field in the LR link, Figure 3. Ultimately, the return will always occur at the end of a production, after a switch has been made to a singleton state, i.e. LL parsing.

At this point the second driver comes into play and looks up the second parse table by LR index for the row number and nonterminal index from the completed end-of-production to determine the location in the grammar code to which control should now be transferred.

Parse tree-stack

It is envisaged that the semantic routines will be responsible for adding semantic records to the parse tree-stack and for building tree structures by combining these, as required, by unlinking them from the stack and relinking them into a tree structure that constitutes the intermediate representation. From there it is possible to convert the representation to assembler code, which may include optimisation. An alternative would be to convert the tree directly into Forth and load that via an optimising compiler.

It is also envisaged that syntax errors will simply generate an abort, with an appropriate error message. Compilers that try to correct errors generally end up with more consequential errors further on in the parse, which only serve to confuse the issue (as in Java). Correcting such errors is very quick on modern desktop computers so making corrections one-by-one easy to do.

7 Rewriting the grammar

Throughout, the intention has been to minimise the rewriting of the grammar. In particular, it has been found unnecessary to rewrite it in a bracket-free form, sometimes called a ‘standard’ form. As shown above, EBNF works fine and can be executed directly in Forth to produce a very compact set of parse tables.

However, there is one area where some rewriting is necessary. This involves the question of Identifier and is because nonterminals that have ‘identifier’ in their names invariably end up at a production that consists of `Identifier` on its own. This means that we get a large number of reduce conflicts – almost fifty in Pascal, including the notorious ‘dangling’ else which occurs in both Pascal and languages based on the C syntax.

Thus, instead of

```
 <field_identifier>    -> Identifier ;
```

we need to write

```
 <field_identifier>    -> FieldIdentifier ;
```

This means that, when a field is defined, its entry in the dictionary must be flagged as being a `FieldIdentifier` token type by the appropriate semantic routine.

8 Conclusion

The scheme given here has found to work very successfully in parsing simple Pascal programs, using a subset of the semantic routines needed for a full compiler. The tables and grammar code used for this are very compact, amounting to around 15K bytes in 16-bit form.

The combination of LL and LR techniques has been found to operate very smoothly. Only 25 LR states are needed in total, so there is no need to try to reduce the very large number of states generated by conventional LR-only parsers using the common SLR and LALR techniques, which also reduce the power of the parser. The parser described here is fully LR-capable. Pathological examples of grammars quoted in the literature, that are not able to be handled using SLR or LALR but still satisfy the conditions for LR parsing, are handled with ease by the parser described here.

For those studying compiler writing or defining a new language, the parser may be set to show how the first and follow sets for the different nonterminals are built up, pass by pass. It is also possible to examine the various productions, including LR links in an easy to read form. A report on the language can also be written automatically as a plain-text file to disk giving a variety of information about the language, including details of the LR states and how they link together. This is reasonable for Pascal which has only 25 LR states.

The report can then be examined at leisure. If needed, the building of the start and follow sets, which require several passes on the grammar code, can also be included in the report.

At the same time, it is still possible to treat the parser as a 'black box' by feeding it with the files required to specify the language syntax without having to do any special coding.

The grammar code and lookup tables are such that the parser can easily be run in any computer environment, although it is simpler to do so in Forth.

References

[1] A.R. Jorden, P.D. Read and I. G. van Breda. Photon counting Reticon system-description and performance. *SPIE Conference, Instrumentation in Astronomy IV*, 331, 368-375, 1982.

[2] I. G. van Breda and N. M Parker. Forth and microprocessor applications at the Royal Greenwich Observatory. *Microprocessors and Microsystems*, 7, 203-211, 1983.

[3] I. G. van Breda and D. J. Thorne. The Handling of Large Format CCD Images. *ESO-OHP Workshop on the Optimization of the Use of CCD Detectors in Astronomy, ESO Conference and Workshop Proceedings*, 25, 83-92, 1986.

[4] C. N. Fischer and R. J. LeBlanc. *Crafting a Compiler.* Benjamin/Cummings Publishing Co., 1988.

[5] I.G. van Breda. Some comments on the Proposed Standard. *EuroForth* 2012.

[6] K. Jensen and N.Wirth, revised by A. B. Mickel and J. F. Miner. *Pascal User Manual and Report.* Springer Verlag, 1991.

Some comments on the proposed Forth Standard

Ian van Breda

Hailsham, East Sussex, UK

Abstract

This paper discusses the current proposals for a new Forth standard following on from ANS Forth. Some changes of name are suggested. It is shown that the proposals for the Extended Characters wordset can either be subsumed into existing definitions for fixed-character sets or be added in a way that make the two schemes compatible. Proposals are also made for record structures coupled with dot notation, that fit well with existing Forth practice and would allow a natural extension to object-oriented Forth.

1 Philosophical

I feel that I am something of an interloper at this conference, as my background in astronomy is rather different from most attendees. I am fully aware that my comments may not always be welcomed by those who have spent an enormous amount of work putting together the proposals for a new Forth Standard [1]. This is even before considering the considerable effort required to reconcile differing opinions, often strongly held and to which I cannot claim to be an exception. My comments are not intended as personal criticism in any way but are based purely on technical considerations. It is to be hoped too that any misconceptions in my comments will be forgiven.

With thousands of Forth users worldwide, establishing a standard is very important. The Holy Grail for desktop computing must be for the computer to run a pre-emptive multi-tasking Forth system on top of which other operating systems can be built, particularly Unix, Mac OS X and Windows.

Providing a standard for Forth presents unique problems due to its extensibility. It must be able to provide a platform for a variety of uses, including robotic instrument control (the original use of Forth), embedded systems and use with modern desktop operating systems.

As operating systems get further and further apart from the hardware, which therefore becomes progressively more difficult to program (such as USB), the plea by the inventor of Forth, Charles Moore, for simplicity in computing becomes ever more valid. Development environments have become very complex and documentation is no longer available in book form; on-line documentation does not have the disciplined structure required of paper manuals and is awash with unexplained acronyms and buzz words, each requiring a link to find out what it means, only to be confronted by further links.

When it comes to hardware, Forth provides a unique capability for accessing hardware directly. In astronomy, this direct access to hardware is especially important, as recent spectacular scientific advances have been almost totally dependent on new technology, both engineering and applied physics. Currently, developments are eagerly awaited in superconducting detector arrays that will revolutionise the observation of the most distant galaxies.

Despite the fact that Forth provides a unique perspective on computing, it is ignored in university courses: there even exists a book, purporting to compare computer languages, that ignores Forth completely (Cezzar [2]). There appear also to be university courses which contain no practical electronics component. We see the results in a myriad of ways in poor programming of a variety domestic devices.

In what follows, where examples are given, they use the truncated form of stack comment used in Forth/68 [3], which is used as a comparison in a number of places. Readers should find no difficulty, however, in interpreting their format.

1.1 Some basic principles

Common practice is not necessarily a good argument for including features in a standard. For example, in the first Forth systems, `LOAD` was used to interpret Forth blocks when compiling programs; the natural extension was to use `LOAD" ..."` for loading named files, yet `INCLUDE` was used instead. Likewise, `S"` was adopted in the ANS Standard for defining a string, when the argument for current practice was extremely weak, stating that some users used `"` and some used `S"`. It is not even clear why some users had departed from using `"` which had been common practice at that point, going right back to the original systems of Charles Moore. What is worse, it even looks rather fussy and ungainly.

The problem with using common practice as an argument is that there is a danger of 'standardisation by stealth'. Whose common practice is it anyway?

Similarly, having always done something wrong is not a good argument for continuing to do so if it is technically incorrect (Section 3.1).

It is also important to take note of developments outside Forth so that, where appropriate, similar techniques can be adopted within the Forth framework.

Giving preference to avoiding breaking of existing code over technical soundness is also not a good argument. For Forth/68, ANS Forth was the first standard Forth considered good enough to move from polyFORTH, which had been used by many programmers up to that point. To make the transition, much code had to be rewritten, i.e. was 'broken' [3]. Nevertheless, the changes were not too difficult to make, as Forth is relatively easy to change and, in particular, to debug.

2 Some general points

2.1 A name for the Standard

The Standard needs a name. The reasons are:

(i) ANS has been deleted in the text for the proposed Standard but programs conforming to the Standard cannot claim to be the only valid Forth programs;

(ii) A name is needed to identify which Standard applies, i.e. a new prefix in place of ANS.

2.2 Cost

Cost of purchase of the Standard needs to be reasonable. ANS Forth is outrageously expensive and does not even include a printed copy, an essential for browsing the Standard.

2.3 Quoting from the Standard

It is important to make clear just what can be quoted form the Standard for those of us who are not copyright lawyers. An email query to ANSI over whether or not the Pascal grammar could be distributed in machine-readable form elicited no reply.

2.4 System name

It is highly desirable to give each system a name, so that conditional compilation can be done depending upon which system is being run at the time. This could be implemented as

`SYSTEM-NAME ( `*`c-addr u`*`)`
Return the address and count for the name of the Forth system.

3 Fundamental Changes

Some changes seem to be essential to the proposals for the Standard.

3.1 Scanning vs. parsing

The terms 'parse' and 'parsing' are used incorrectly almost entirely throughout the text for the Standard.

In both the treatment of natural language and the theory of computer languages parsing implies analysing the structure of a sentence or language construct. In computing it refers to checking that the tokens extracted from the source code of a program follow the grammatical rules of the language [3].

The correct term, as used elsewhere in computing, is 'scan', which is the process of extracting language tokens from source code. This suggests that the vast majority of the descriptive text in the proposed Standard needs to be changed from variants of 'parse' to their 'scan' equivalents.

We cannot change `PARSE` itself, as it is now cast in stone (the penalty for inaccuracy in setting up the ANS Standard). However, it is possible adopt `SCAN-NAME` in place of `PARSE-NAME`. It would also be useful to have a multi-line `SCAN`, possibly similar to that in Forth/68, which bypasses comments.

The argument that it has always been done this way is surely not valid in the face of the fundamental usage of the term, parsing, elsewhere: Forth cannot ignore the outside world and, at least in this case, it is irrefutably common practice to use scanning for this type of process.

3.2 Word lists and vocabularies

Word lists were poorly designed in ANS, despite the intention of producing a set of primitives that could be used in a variety of schemes.

The reason for the problems with the Standard is that there are unnecessary hidden effects on the search order. The committee indeed accepted that the implementation of `ALSO` and `ONLY` with their hidden and not very intuitive effects was controversial.

It is difficult to see how this scheme was accepted for the Standard. The way it should have been is

FORTH (-- *wid*)
Return the *wid* for `FORTH`.

ONLY (*wid* --)
Given a *wid*, make it the only word list in the search order.

ALSO (*wid* --)
Add the *wid* argument to the top of the search order.

DEFINITIONS (*wid* --)
Set the compilation vocabulary to *wid*. Add *wid* to search order if not already at the top. Options on search order are to leave it as is or set it to what it was when the vocabulary was defined.

VOCABULARY (*$* --)
Define a new named word list that returns its *wid* on execution.

With named vocabularies, which (fortunately) ANS avoids, it is possible to set up a specific search order without using `ONLY`/ `ALSO`. In EBNF

```
ENDVIA { <vocab-name> } ENDVIA
```

The search order is set to the list of vocabulary names, last named to be searched first.

Even though the ANS Standard defines `CODE` and `;CODE` which provide links to assembler code, it is acknowledged that the actual way that the assembler works is system-dependent, as it must be on different processors. For this reason, `ASSEMBLER` should have been left out of the standard. In Forth/68, switching to assembler code sets `ASSEMBLER` as a transient at the top of the search order, which is discarded whenever a new definition is added to the dictionary or `ENDCODE` is executed.

Had the above scheme been adopted in ANS (and would surely have been less controversial) it would have made `FORTH-WORDLIST` redundant and, maybe less so, `SET-CURRENT`.

Forth/68 uses the above scheme, for example, in the MetaForth cross compiler used to compile the minimal kernel for the system and for the Pascal parser [3], which makes extensive use of vocabularies. There is a software switch for a change of mode to ANS-compatible, although it is never used in practice.

Fortunately, it is possible to make an ANS Standard program compatible with the above by using very simple redefinitions.

It is unfortunate that the treatment of word lists has been cast in stone. While I would like to see the ANS scheme deprecated and replaced by the above, it is probably impractical to do so. However, it does highlight the dangers when what was technically a poor choice was made for the Standard.

3.3 LOCALS| and {:

`{:` is a big step-up from `LOCALS|` not least because the number of local definitions permitted has been doubled, though it is not clear why it needs to be limited at all: the original eight seems to have been determined by the number of D-registers in an MC680xx processor, so that only two instructions were needed to transfer locals from the parameter stack to the return stack.

Also, it is not clear just why it should not be possible to access locals within, say, an `>R ... R>` segment after `LOCALS|`. For systems that place locals and loop parameters on the return stack, return-stack tracking must be used anyway, so there is no problem with access to locals. Likewise, if the locals are placed elsewhere, there is no problem anyway.

There is an inconsistency in the wording of the proposed Standard: in 13.3.3.2 d) it is implied that the return stack can be manipulated after `LOCALS|` variables have been defined, so long as return-stack balance is maintained; A13.3 implies that the return stack cannot be manipulated at all at this point.

The ability to include uninitialised variables in a `{:` list is a big plus, as it does not involve any transfer from the parameter stack.

However, the impression given in the Standard proposal is that `{: ... :}` would be used in place of the parenthetical comments normally used at the front of a definition. This is both very limiting and surprising, given the large number of named variables allowed. In Forth/68 `LOCALS|` are invariably preceded by some stack manipulation and `{:` would be used in the same context. In this way of using it, the `-- <out>` becomes irrelevant. In this system, the stack arguments are given in short form at the front of a definition, while the local names are more descriptive, see [4]. For example, we might use *n-indx* as a mnemonic in a stack comment but `grammarIndex` for its more descriptive `LOCALS|` name.

If large numbers of locals are to be allowed, or even relatively few descriptive names are to be used, it is essential that `{:` segments be able to cover more than one line; the same goes for `LOCALS|`.

The following changes are therefore suggested:

- Use the same order on the stack as with `LOCALS|`. Reasons are, firstly, that the Standard must be technically consistent. Arguments that the proposed order is common practice do not hold water here, as it is admitted that there are several ways of implementing the `{:` function.

  Secondly, this will 'break' existing code for those that use different schemes anyway, so that argument does not apply.

  Thirdly, if items are added to the stack as commonly happens before the locals are defined, it is easier to track backwards through the stack to identify the local names.
- The `-- <out>` should be dropped, as it is irrelevant to the function of `{:`, unless the Standard specifies that `{:` is to be used *only* as a replacement for the normal `( ... )` argument comments at the front of colon and `CODE` definitions; this would betray a very limited view of the capability.
- Given the large number of possible locals, it essential that both `LOCALS|` and `{:` be allowed to cover more than one line.

4 Code vectoring

It is much more common elsewhere in computing to use the term vectoring for the capability that `DEFER` is intended to provide.

All references that I have been able to find to deferring of code refer to deferring of the *time of execution* of the code, for example in Java, where a horribly complex way of doing this is needed. Of course, in multi-tasking Forth, this is both much easier to implement and read.

However, it is suggested that the correct term to use is vectoring, which is in common use. For example it has been used for many years for handling of interrupts, and is even given conceptually in MVP-FORTH where an example is shown of how to do this for `PAGE`. This concept has also been in use in Forth/68 for for many years for the `TYPE`-style routines, `TYPE`, `PAGE` and `CR`, as a group, also for error message generation for routines in the kernel, which are needed before support for error-message display has been loaded.

It is therefore suggested that `DEFER`, `DEFER@` and `DEFER!` be replaced by

`VECTORED, VECTOR@ and VECTOR!`

This has the advantage that `VECTORED` does not get confused with `POSTPONE`, which is easy to do with `DEFER`.

In this respect it is useful to have a definition

NULL
: Take no action. This is the equivalent of the usual `NOP` in assembler.

It is useful as a default for vectoring to avoid crashes if the vectored code is called accidentally before the execute token has been set up.

5 Strings and characters

Given that there is very little provision for string handling in the Standard proposal, it is something of a surprise that so many of the proposed changes to the Standard involve characters, strings and key strokes.

For those of us who do not normally use such extensions, the Standard needs to provide much more explanation as to how the various changes are to be used and why they should be included in the Standard, rather than being treated as specific to particular applications.

Again, it is not at all clear why operations such as `REPLACES`, `SUBSTITUTE` and `UNESCAPE` should be in the Standard rather than part of an application. They seem to belong in very specialised applications.

5.1 String operations

Some suggestions for very basic but highly desirable extensions for string support are:

CATENATE (*c-addr*$_1$ *u*$_1$ *c-addr*$_2$ *u*$_2$ *u*$_3$)
: Append the first string the end of the second but limiting the maximum length to u_3. This is particularly useful for putting together file path-name strings.

STRING-COPY (*c-addr*$_1$ *u*$_1$ *c-addr*$_2$ *u*$_2$)
: Copy a string across to a buffer, truncating the length to u_2 if u_1>u_2.

COMPARE-TEXT (*c-addr*$_1$ *u*$_1$ *c-addr*$_2$ *u*$_2$ -- *n*)
: This is a case-insensitive version of `COMPARE`. It can be used for putting definition names in alphabetical order in `WORDS` listings but is also useful in other sorting situations.

These can easily be defined to work seamlessly with the Extended Character set.

5.2 Characters and bytes

There is a problem with the ANS Standard which seems to have been written to allow Forth systems to use character coding of fixed but unspecified length. It is presumably for this reason that `CHAR` has been used instead of the somewhat clearer ASCII.

However, Conklin and Rather [5] specify that `C,` should append a byte to the data space, which does not accord with the Standard. Similar considerations apply to `CMOVE` and `CMOVE>`, `C@` and `C!`. This assumption is made in several places in the text, despite the claim at the front of the book that it 'features Standard Forth'.

The limitation of a maximum of 255 characters in strings implies the use of `C@` to extract the length of a string for ASCII-based systems. Conklin and Rather make this explicit in a definition that they give for `COUNT`. However, this can all be generalised very neatly using definitions similar to those for vectoring, Section 4. Thus

COUNT (*c-addr*$_1$ -- *c-addr*$_2$ *u*)
: This is the same as the ANS Standard definition, which returns the address/count for a string. The generalisation of this allows for the fact that strings might be implemented as, say, 16-bit counts with 8-bit characters. Also it allows for Extended Character strings, which might also have a byte length for the string following the count and which would be hidden from the user. The only requirement would be that the count itself would need to be character-aligned.

COUNT@ (*c-addr* -- *u*)
: Fetch the string character count stored at the given address according to the system-dependent string count size.

COUNT! (*u* *c-addr*)
: Store the string character count *u* at the given address according to the system count size.

This means that, to match the Standard, we need operations equivalent to `C@` etc. but that are in sizes that match the address units of the computer. It is suggested that these be prefixed by the letter 'B', which reflects the fact that the majority of processors use byte addressing (although this is not required):

`B@ ( addr -- u)`
: Fetch the item that is one address-unit wide stored at the address, unsigned.

`B! ( u -- addr)`
: Store *u* at the address, truncating its value if it is larger than the maximum unsigned number that will fit in one address unt.

`B+! ( u addr)`
: Add *u* to the address-unit value at the address, truncating the result if it is larger than the maximum unsigned number that will fit in one address unit.

While character operations may continue for bytes, their use should be deprecated and replaced by their B-equivalents for programs following the new Standard.

With this notation, `MOVE` should really have been `BMOVE`. However, accepting that it is an address-unit move, it needs to be generalised so that it can move bytes either up or down when the source and destination lists overlap; it is easy enough to implement this, as it only involves a check on the source and destination addresses. For speed, we need also to have a cell-aligned move.

`MOVE ( `*addr*$_1$ *addr*$_2$` -- u)`
: Move *u* addresss-unit values from *addr*$_1$ to *addr*$_2$. The list of values may be moved to upper or lower addresses and may overlap.

`CELL-MOVE ( a-`*addr*$_1$ `a-`*addr*$_2$` -- u)`
: Move *u* cell values from *a-addr*$_1$ to *a-ddr*$_2$, with addresses aligned. The list of values may be moved to upper or lower addresses and may overlap.

5.3 Extended Characters Word Set

In what follows, it is assumed that the extended characters are used as the main input/output for the Forth system, rather than occasionally interpreting a file that uses the Extended Character Set. This is implicit in the Standard proposal, which indicates that `TYPE` and `ACCEPT` need to be generalised. The same must also apply to such definitions as `."`, `S"`, `C!` and `WORD`.

However, if different codings are to be supported on a given system, it is a simple matter to use code vectoring to switch between the character codings.

If we assume that the new definitions will also be found useful in systems using fixed character-sizes, then it is a simple matter to provide definitions that will work either within a fixed character-size system or one that uses the Extended Character Set. This means that X can be dropped as a prefix for the definitions.

As with the definitions mentioned above, most of the X-definitions can simply be generalisations of existing definitions. In particular

`XHOLD, XCHAR+, XEMIT, XKEY, XKEY?, EKEY>XCHAR, XC,`
`--> HOLD, CHAR+, EMIT, KEY, KEY?, EKEY>CHAR, C,`
: These are straightforward generalisations of existing definitions.

`XCHAR-, XC!+, XC@+ --> CHAR-, C!+, C@+`
: If these definitions are found useful for the Extended Character set, then equivalents are easy to define for fixed-width character sets.

`XC!+? --> C!+?`
: This can again be implemented in fixed character-size systems, although it may be trivial in such cases.

`+X/STRING, X\STRING- --> +/STRING, \STRING`
: These have obvious meanings in the context of fixed-width encoding but there are queries over the meaning of the stack arguments (see below).

`X-SIZE, X-WIDTH, XC-WIDTH --> CHAR-SIZE, STRING-WIDTH, CHAR-WIDTH`
: All these have obvious meanings in fixed-width encodings and might be considered redundant but there is an advantage in being able to use the same definitions in different coding systems.

The above approach is much neater than the proposed Standard in that they are either generalisations of existing standard definitions or are definitions that can easily be applied to fixed character-size codings.

5.3.1 Buffers

In several of the definitions for the Extended Character Words, the term 'string buffer' is used. While the address of the buffer is reasonably clear, the meaning of the count/size argument is not. For example `XC!+?` has arguments `(` *xchar xc-addr*$_1$ *u*$_1$ `--` *xc-addr*$_2$ *u*$_2$ *flag*`)` but u_l is not defined. Is it the length of the buffer? If so, where is the character supposed to go? Is there a hidden size or place within the buffer? Generally, at least two parameters are required in such circumstances: some measure of the size of the buffer and the size of any string currently within that buffer.

In `+X/STRING`, the length parameter appears to refer to the buffer size rather than a string within it, while `-TRAILING-GRABAGE` seems to refer to a string as such.

In `X\STRING-`, the description refers to the 'penultimate' character when it appears to be that the last one that is the only character that is relevant.

5.4 K-definitions and EKEY

Again it is difficult to see what relevance many of these definitions have in the context of an event-driven system: the function keys need to be requested explicitly and do not initiate any action of themselves.

However, there is a problem with this on the Apple Mac, where the operating system uses several of the function keys to modify the way in which windows are displayed. Even though Forth/68 intercepts the keyboard interrupt, OS X does not give any output on `EKEY` when these particular keys are struck.

There is a similar question over user aborts, which are an essential for computers that control robotic scientific instruments. Here the query is: should the user be able to generate an abort when `EKEY` has been called? In Forth/68, a user abort is generated for Command-Escape and is much easier to implement if it is allowed to generate an abort on `EKEY`.

Given the problems with function keys, it is suggested that the Standard include a phrase for `EKEY` that permits response to control keys to be implementation-dependent, especially as it may not be able to respond to the function keys. `KEY` is slightly different but does need to include a similar comment that the user may generate an abort after `KEY` has been called, at which point the abort takes preference.

5.5 Pausing dumps

The ANS Standard rejected the use of `X-ON/X-OFF` (ctrl/Q amd ctrl/S, respectively) ostensibly because different systems respond differently to these characters, even though it is heavily slanted towards ASCII and these control characters are considered to be a 'standard' use of ASCII coding. `X-ON` and `X-OFF`, or equivalents, are an essential part of a programmer's toolkit and should be included in some form in the Standard. System-dependent coding for pausing dumps would nevertheless be entirely satisfactory.

Similarly, for pe-emptive multi-tasking systems at least, some sort of abort key should be provided to generate a user interrupt, again system-dependent.

6 Records and other structures

The Standard suggests use of `BEGIN-STRUCTURE` and `END-STRUCTURE`, presumably taken from C (the hyphen is inconsistent with `ENDCASE` and `ENDOF`). However, since an object might reasonably also be considered a structure, a better name might be `RECORD`, as used in Pascal, coupled with `ENDRECORD`.

We can then follow what is similar to common practice elsewhere by making `RECORD` define a new word list to which to add field definitions, in a fashion similar to that given in the Standard proposal.

Thus we might have

`RECORD ( `*`$ -- addr 0`*`)`

This expects the name of a new record-type and assigns a new word list, making that the compilation word list. It leaves the parameter field address for the named record on the stack and an initial zero field offset.

`FIELD DEFINERS`

These take a form similar to that in the Standard proposal and place definitions in the wordlist/vocabulary for the record:

```
        INT: defines a new (32-bit) integer field;
        WORD: defines a 16-bit unsigned integer field;
        BYTE: defines an 8-bit unsigned integer field;
        CHAR: defines a character field;
        SFLOAT: defines a single-precision floating-point field;
        DFLOAT: defines a double-precision floating-point field;
        (n) STRING: defines a string field;
        n FIELD: defines an uncommitted field of length n;
        <record-type> RECORD: defines an embedded record.
```

As each field is defined, the offset is first aligned in accordance with the data type and then incremented according to the data-item size. Each field definition includes a data size and has its offset within the record as a parameter.

There are questions over the string field. Options are to use an in-line counted string or a handle for a relocatable block containing a string. This is best left for discussion elsewhere.

`ENDRECORD ( `*`addr offset`*`)`
Terminates the record structure by setting the data size for the record from the final field-offset.

6.1 Record Instances and dot notation

If `RECORD` defines a new record *type* that, when executed, returns the data size and word-list identifier for the fields in the record, then we can create an instance of the record in, say, Forth data space (e.g. the dictionary). We can then define an instance of the record:

```
  <record-type> NEW-RECORD <instance-name>
```

When `<instance-name>` is executed, it returns the address of the instance data, along with the word-list identifier for `<record-type>`. The *wid* can be on the parameter stack, though there is a strong case for placing it in an (anonymous) user variable, as it never needs nesting. If, in addition, field names simply add their offset to whatever is on the stack, the way is open to use dot notation. For example,

```
  <instance-name>.<field-name>
```

can be split into two tokens, each of which is processed separately.

In execute mode, the first token is executed and returns the record-instance address plus *wid*. The next token (i.e. field) is looked up in the word list and executed so that the address of the field is returned. If compiling, an address literal must be assembled that returns the address on the stack. This is very straightforward for systems using direct-execution/subroutine-threading in colon definitions but would need a little more thought in other forms of threading.

This would allow the use of forth accessors and setters for the fields, including `TO`.

An advantage of using a *wid* hidden inside a user variable is that we can write

```
  <instance-name>
```

simply to return the address of the record without having to pop the *wid*, which then remains hidden away and unused. This would work in both execution and compilation modes.

The above can be used with a new word, `FROM`, as

```
  FROM  <instance-name>.<field-name>
```

which would push the contents of the field onto the parameter stack. `FROM` is very similar to `TO` and would be system-dependent in the way it was implemented. For subroutine-threaded systems, this could be implemented as an assembler push to the stack, either as an address or a field value, allowing for the size of the appropriate field, e.g. extending character values to, say, 32-bit.

`TO`, with slight generalisation, could be used in the same way:

```
  TO  <instance-name>.<field-name>
```

In both cases, the size of the data field would need to be accessible, which would modify the way in which the token processing took place but that happens anyway in the case of a `(2)VALUE` variable.

6.1.1 Object-oriented programming

Considerable interest has been shown in object-oriented programming at this conference, see Ertl [6] Schleisiek [7] and Haley [8]. The above can then readily be extended for this purpose. While this is not the place to put forward a full OOP proposal for Forth, the dot notation follows that used in C, Java and Delphi and is both technically sound and easily implemented in Forth.

For OOP, we could use `CLASS`, in place of `RECORD`, to define a root class and `SUBCLASS` to define a subclass of another class, given its name. `FROM` and `TO` could then also be applied to objects, whether they are addressed directly or, perhaps more usually, through a handle, in a very Forth-like way and would integrate well into existing Forth techniques for handling stack data.

A couple of points might be noted. Firstly, the distinction between *private* and *protected* class members is often over-egged by OOP followers. It is possible to get close to this by using `GLOBAL` and `LOCAL` definitions, as described in Section 8.5. *Private* can be implemented by calling `LOCAL`, *public* by calling `GLOBAL`, with `INVISIBLE-TO` at the end of the file removing the private definition names from the dictionary. Subclasses for the given class can access private members if they are defined within the same file. There would then be no equivalent for protected members accessible to subclasses defined outside the confines of the file (Forth/68 could implement all three categories but the above scheme is simpler). In any event, having three levels of visibility is only a problem for bad programmers, which surely we don't have in Forth!

Access to inherited members, particularly methods, requires access to superclasses. This is not too difficult to do but would be system-specific in its implementation.

6.1.2 Advantages

By using dot notation, coupled with `TO` and `FROM`, we have a scheme that not only conforms to existing practice outside Forth but, at the same time, dovetails very neatly into the existing ways in which stack arguments are processed using existing Forth operators. It is also simple to understand and elegant to use. It is easy to implement in Forth in such a way that what the Forth programmer sees is system-independent, apart maybe from deciding on sizes of the integer fields.

It also has the added advantage, not possessed by the current `BEGIN-STRUCTURE` proposal, that field names are attached to particular record types and so can be used freely for definitions elsewhere in an application, including fields in other records. The scheme also provides a natural progression from support for records to the implementation of object-oriented programming.

7 Modern developments

Since most Forth systems are run on modern desktop operatings systems, some attention needs to be paid to these.

7.1 File handling

When including files, it is essential to be able to use both full and partial file names. For example in the file that is included when processing the LR parser described in [3], the string

```
S" ::Compilers:Pascal:Character Classes"
```

is used for loading the file containing the character classes for Pascal. This involves backing up by a couple of folders from that containing the Forth system, then coming back down through the Compilers and Pascal folders to reach the required file. In Mac OS Classic, this is achieved using colons.

It is necessary to standardise the way in which these strings are expressed so that it is possible to open files using the following:

(i) a full path name starting at the root directory;

(ii) a parth name relative to a directory backed up from the one in which Forth resides;

(iii) a simple name for a file located in the same directory as the Forth application.

The Apple Mac only needs one character to cover these requirements.

No case is made here for following Apple Mac practice. Rather, a standard way is required that allows path names to be converted to a form that works on a particular system.

Use of wildcards, as in Unix, could provide a considerable challenge on some operating systems and it seems unlikely that the Standard should attempt so support this way of specifying file names.

An important feature of GUIs is the ability to open a file using a system-dependent dialogue. It is therefore important for Forth to provide a standard means of bringing up such a dialogue.

`GET-FILE ( u-mode -- fileid ior)`
: Open a dialogue that opens a file by name using the read/write mode on the stack and return its *fileid*, zero if the user cancelled, and an io-result, zero if no error.

`LOAD-FILE`
: `INCLUDE` a file selected from the OS standard file dialogue. `INCLUDE-FILE` is already in use in the ANS Standard, so cannot be used here.

`INCLUDE` and `REQUIRE` cannot be used as defined, since file (and directory) names often contain spaces. Also file names can start with spaces, which are significant. Instead something like `INCLUDE"` is needed.

`INCLUDE"`
: Scan the input stream for a string delimited by a double quote. Open the file whose name is specified by the string, which can be an extended path name, and proceed as specified in `INCLUDED`.

`REQUIRE` has the same problem for file names as `INCLUDE`. However, there is potentially a fundamental flaw with `REQUIRE` in that the files are not necessarily loaded in a specified order so, if there is a name clash with definitions loaded in other files, there will be an ambiguity over which definition is the one that should be used. `REQUIRE` and `REQUIRED` therefore really need to be abandoned. Note that a close equivalent, `Uses`, in Delphi still requires the 'unit' names to be given in their correct order.

Also extremely useful for examining the contents of files is

`DUMP-FILE ( wid ud-start u-size)`
: Display the contents of a file in the normal `DUMP` format starting at location *ud-start* covering *u-size* address units

7.2 Relocatable blocks

Support for relocatable blocks is provided by the operating system for desktop computers, by means of handles, i.e. indirect addresses, for access. This allows the size of the a block to be changed, moving the block when necessary. For embedded systems, this can be accomplished by placing the block in the Forth data space and moving it explicitly when the size is increased but, again, it is necessary to use indirect addressing.

Relocatable blocks are very useful for structures whose size is not known beforehand, particularly for stacks and queues. Forth/68 uses relocatable blocks for these purposes, especially dictionary headers, with control parameters placed at the start of each block. Stacks and queues may use fixed or variable sizes of data items and are automatically extended in size, when needed.

At the very least, basic definitions are needed for relocatable blocks:

`NEW-HANDLE ( u -- hndl rslt)`
: Allocate a relocatable block of size *u* address units. Return a handle and a result code that is zero if okay, non-zero if it has not been possible to allocate a block of the required size.

`FREE-HANDLE ( hndl -- rslt)`
: Dispose of a relocatable block, given its handle. Return a zero if the operation was performed correctly, non-zero if there has been an error, e.g. if the input argument is not a valid handle.

7.3 Menus

Menus are an essential part of GUIs and, indeed, had been used with RS-232 video terminals at the Royal Greenwich Observatory (RGO) well before Mac OS and Windows started to use them.

It is therefore important to have a standard scheme so that menus can be transportable. At present it is necessary to include a separate menu file for each system that a program is to be run on. Forth/68 makes use of the primitive definition, `ENTER`, to enter names into the dictionary. As this expects a string address/count, it can enter names that include spaces. Each name within a menu is defined in a vocabulary that belongs to that menu, so the names can be looked up in the dictionary in a robust manner.

For example, in an application for renumbering digital images by capture date, in an `Actions` menu:

```
VOCABULARY Actions  Actions DEFINITIONS
S" Tag Movies..."   :ENTER   TAG-PREAMBLE  RENUMBER-FILES ;
```

where `ENTER:` is the 'colon' version of `ENTER`.

No case is made here for using this scheme in the Standard, although it is easy to implement. However, a standard method of setting up menus is highly desirable.

8 Miscellany

8.1 Interpretation Semantics

Some definitions don’t have interpretation semantics but have very obvious behaviour in that situation.

`EXIT`

This is very useful during development for exiting from compiling a file.

`."`

`."` has an obvious meaning for interpretation and is useful for displaying progress of a compilation. It also looks more elegant in a source-file when displaying text during compilation than the rather ugly `.(` in the ANS Standard, which is redundant anyway if `."` is allowed to have interpretation semantics.

`S"`

Although S" has interpretation semantics in the File-Access Word Set, it does not in the Core Word Set. Essentially, this makes the two versions incompatible and is somewhat confusing. It is therefore suggested that the two versions be given the same interprettion semantics.

It is not clear why the ANS Standard adopted `S"` in favour of `"` since the latter had been used since Charles Moore's original Forth came out and surely could have been able to claim common practice at the time.

8.2 UNLOOP

In the ANS Standard, `UNLOOP` is a somewhat enigmatic definition in that it must be followed by either `EXIT` or `UNLOOP` itself and it is not clear what happens if neither of these follows.

However, there is a very useful feature if it can be followed by `LEAVE`. If used inside an inner nested loop, `UNLOOP LEAVE` would perfom the task of leaving the outer loop and would branch past its `LOOP` terminator.

More than one `UNLOOP` could be used in a similar way. This avoids what can be a somewhat complicated way of getting past the `LOOP` terminator in an outer `LOOP`.

8.3 EMPTY

`EMPTY` is essential in multi-tasking systems for emptying a terminal task dictionary, as it includes setting of defaults, such as search order, and closing any files opened by a previous application. These are functions that cannot be undertaken using `MARKER` which, in any case, would always need to leave a marker at the start of each terminal task.

8.4 Stack pointer access and SP@

Accepting that not all systems will have access to the stack pointer via `SP@`, it is nevertheless very useful on desktop and similar systems, where it will normally be available. This allows data structures placed on the stack to be accessed, an important feature if other languages are also supported. The fact that the stack is not visible on all systems should not prevent `SP@` from being included in the Standard as an option.

In this context, it can be useful to have a definition that reverses the stack order, especially in those systems where the stack pointer is not visible:

`>STACK< ( u)`
: Reverse the order of the top *u* items underneath the argument on the parameter stack.

8.5 WORDS

To be usable, `WORDS` really needs the list of definition names to be able to be displayed in alphabetical order. For example, in MacForth, `WORDS` displays the definition names in chronological order, making it very difficult to see what words have been included in the dictionary. The problem is made worse if all words used in building the system are included in the list, as many of these are of no interest to the user.

Forth/68 includes some useful definitions that have been in use for many years and which discard the headers for selected words:

`LOCAL`
: This is a compiler directive that causes subsequent definitions to be marked as local for discarding later.

`|`
: Makes the next definition `LOCAL`, then retruns to the `LOCAL/GLOBAL` mode current at the time.

`GLOBAL`
: Causes subsequent definitions to be marked to be kept visible.

`INVISIBLE`
: Removes the headers for all definitions currently visible in the dictionary that have been marked as `LOCAL`. There is also an `INVISIBLE-TO` which makes `LOCAL` definitions back to a particular point invisible.

This means that the many field names for certain data structures, subroutine labels and Mac OS 9 assembler traps in Forth/68 can be discarded to help reduce dictionary 'entropy'. They therefore do not appear in `WORDS` listings.

It is important to note that the purpose here is not to reduce memory usage but instead to simplify the dictonary.

9 Number conversion

The proposals for converting numbers are very complex. Certainly, these numbers would look more elegant if the discriminator followed the sign.

I would also suggest using the prefix `0x` or `0X` for hexadecimal numbers as used in C. I have used this form for many years (common practice?). Other numbers, including octal and binary were usually accomplished simply by base-switching, since they are only used occasionally. It is not at all clear that all these different forms are really necessary.

10 Drawing tools

Since the Standard includes `AT-XY` for character-based positioning, it would seem desirable to include definitions for drawing simple lines. Forth/68 has

`MOVE-TO ( u-h u-v)`
: Moves the 'drawing pen' to a position in the output window, *u-h* pixels in the horizontal position and *u-v* vertical pixels, measured from the top left of the drawing area of the output window.

`LINE-TO ( u-h u-v)`
: This is similar to `MOVE-TO` but draws a straight line from the current position to that specified.

Forth/68 also has a definition that allows the output font to be set by name, `FONT"`; its size in pixels/points is set by `PT`. Other definitions allow the fore-colour and background colour to be set to some basic colours such as `BLACK`, `RED` and `MAGENTA`. The fore-colour specifies the colour that will be used for drawing lines and text, while the background colour specifies the colour for the general background.

11 Execution timing

It is often useful to be able to compare program execution times on different systems. A standard means of accessing a clock is needed so the execution speeds can be compared when the same definitions are run on different systems. Resolution does not matter too much but does need to be much less than one second. Mac OS 9 has a system variable called `'Ticks'`, that is measured in 60Hz clock ticks and is generally adequate for this purpose, although a resolution of one millisecond is preferable.

12 Aborting

In the control of what has come to be called robotic instruments, it is important to be able to abort in a controlled fashion. This is also generally useful if there is a chance that a definition will end up in an infinite loop.

One of the first requirements found with the RGO microdensitometer, [3], was the need for both pre-emptive multi-tasking and the availability of a controlled user abort from the keyboard. The practical need for this was that, if a scan were aborted without switching off the motors, these could run onto end stops and required manual rewinding of the drive screws, resetting of the power supplies and loss of registration of the carriage position, which could also mean having to discard any scans made previously. The multi-tasking had to be pre-emptive to ensure that the system would respond fast enough to the abort key.

The solution adopted was to implement an `ESCAPE...ENDECAPE` structure at the start of any definition that needs a controlled abort exit. For example, if `motor` were a `LOCALS|` variable to identify a motor that was being switched on and off, the phrase

```
ESCAPE  motor OFF  ENDESCAPE   motor ON ...
```

could be used. Although `CATCH/THROW` can potentially do this, it is not straightforward in use, the principal reason being that handling an exception is split between two definitions, making the caller responsible for an abort generated from within the called routine (including the case where the user might have pressed the abort key).

The `try...catch` mechanism, as implemented in languages, such as C^{++} and Java does put the `catch` in the same routine but the `catch` only catches specific exceptions.

The `ESCAPE` mechanism is probably the easiest to use. Firstly, it occurs in the routine where the exception is generated. Secondly, in Forth/68, the `THROW` code for an exception can be accessed through a user variable, `THROW-CODE`, so that special action can be taken for any given type of exception. However, this is not proposed for the Standard.

Since `CATCH` has been used in ANS Forth in a way that is different from other languages, it is suggested that the `TRY...FINALLY...ENDTRY` mechanism, similar to that in Delphi, be adopted for the Forth Standard. The above code would then be written as

```
TRY  motor ON ...  FINALLY   motor OFF  ENDTRY
```

The code following `FINALLY` is always executed at the end, whether or not there has been an exception. There are two options on the way in which `FINALLY` works. The first would be for it to return the `THROW` code on the parameter stack, in a manner similar to `CATCH`, zero if there has been no abort; the second would be for it not to return anything but to provide a user variable, like `THROW-CODE` above, which could allow different exceptions to be handled individually or, indeed, allow different actions depending upon whether or not there had been an abort.

Big advantages for this mechanism over `CATCH` are:

- Crucially, the exception handler occurs at the 'heart of the action' and is handled at the same level as the point at which the exception occurs;
- Code following the `FINALLY` follows on naturally from the code before it;
- It is not necessary to access the code before the `FINALLY` with the somewhat ungainly `[']`;
- Any `LOCALS|` words can be used within the `FINALLY` code, including when there is an exception – these are not accessible with `CATCH`;
- At least in robotic instrument control, it is likely that the code following the `FINALLY` will be executed whether or not there is an exception.

References

[1] Forth Standards Committee. *Forth 200x Draft 11.1*. 29th February, 2012

[2] R. Cezzar. *A Guide to Programming Languages: Overview and Comparison*. Computer Science Library, Artech House. 1995.

[3] LG. van Breda. Building an LR parser for Pascal using Forth. *EuroForth* 2012.

[4] S. Pelc. Notation Matters. *EuroForth* 2012.

[5] E.K. Conklin and E.D. Rather. *Forth Programmer's Handbook*. FORTH, Inc. 1998.

[6] M.A. Ertl. Objects 2. *EuroForth* 2012.

[7] K. Schleisiek. Explaining simpleOOP. *EuroForth* 2012.

[8] A. Haley. Standardise Forth OOP. *Euroforth* 2012

Methods in objects2: Duck Typing and Performance

M. Anton Ertl*
TU Wien

Abstract

The major new feature of objects2 is defining methods for any class (like in Smalltalk): this means that we can have two classes that are unrelated by inheritance, yet react to the same messages and can be used in the same contexts; this is also known as duck typing. This paper discusses the implementation of method dispatch for these general selectors as well as the more restricted class selectors of the original `objects.fs`, and compares the memory and execution time costs of these method selector implementations: Unhashed general selectors are as fast as class selectors (down to two instructions), but can consume a lot of memory (megabytes of dispatch tables for large class hierarchies); hashed general selectors are significantly slower ($\geq$ 43 cycles), but consume less memory. Programmers don't need to choose a selector implementation up front; instead, it is easy to switch between them later, on a per-selector basis.

1 Introduction

My `objects.fs` package provided Java-inspired facilities for defining methods: Essentially the programmer defines a method selector for a certain class or interface, and can then only define methods for this selector for descendent classes of that class, or (for interface selectors) for classes implementing this interface.

One disadvantage of this approach and the way it was implemented in `objects.fs` was that passing an object of the wrong class to a selector was not detected. Detecting this would be useful for debugging, and also useful for, e.g., implementing proxies that pass on every not-understood method call to another object.

Also, some people argued that Smalltalk-style methods, which can be defined for any class, would be useful. They would allow the use of *duck typing*: A type is defined as a set of selectors, and every class/object for which methods are defined for these selectors, has this type.

*Correspondence Address: Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria; `anton@mips.complang.tuwien.ac.at`

At first the implementation of these features seemed to me too expensive in run-time, and the benefits did not appear to be very significant. But eventually I learned about more efficient implementation techniques as well as additional uses for these features, so I set out to devise objects2, which provides these features (as well as backwards compatibility with `objects.fs`).

In this paper I look at the basic syntax (Section 2), at various method dispatch techniques (Section 3 and their performance (Section 5), and at the minimally invasive ways offered by objects2 for selecting between dispatch techniques (Section 6). It also discusses (Section 4) how to implement the current object pointer and measures the resulting execution time (Section 5).

What this paper does not discuss whether you should use Smalltalk-style methods and duck typing or use than Java/C++-like methods. Objects2 gives you both options (with a very easy transition between them, and the choice available per selector), and it is up to you to decide which one you want or need to use. This paper also does not give a general documentation of objects2; the documentation comes with the package.

2 Defining methods

Figure 1 shows an example program that defines three classes: A, it's child A1, and the unrelated class B (apart from the common ancestor class `object`, which is unavoidable in objects2).

It also defines two method selectors: `foo` and `bar`; there are two method definitions for each of these selectors. The first method definition for a name defines the selector (a Forth word with that name), any further method definition just defines the method (an anonymous colon definition) and makes it the method that the selector calls for the current class and it's children.[1]

The `some-A1 foo` call demonstrates that A1 inherits the foo-A method from A. The `some-A bar` example demonstrates that objects2 reports if a selector is invoked for a class for which no method is

[1]This kind of conditional definition is very unusual in Forth; it is due to the fact that we we want to optionally use duck typing, so we don't want to have to define the selector beforehand, as was done in `objects.fs`).

```
object class
:: foo ." foo-A" ;;
end-class A

A class \ child class of A
:: bar ." bar-A1" ;;
end-class A1

A  heap-new constant some-A
A1 heap-new constant some-A1

object class
:: foo ." foo-B" ;;
:: not-understood ( sel-xt obj -- )
  ( sel-xt ) some-A1 swap execute ;;
end-class B

B  heap-new constant some-B

some-A1 foo \ prints foo-A
some-B  foo \ prints foo-B
some-A  bar \ method not understood
some-B  bar \ prints bar-A1
```

Selectors	Classes: object	A	A1	B
not-understood	nu-obj	nu-obj	nu-obj	nu-B
foo	udfoo	foo-A	foo-A	foo-B
bar	udbar	udbar	bar-A1	udbar

Figure 1: An example program and its class×selector matrix

defined for the selector (in contrast to `objects.fs`, which would just blindly try to `execute` some xt, with unpredictable results).

Finally, the `some-B bar` call demonstrates the `not-understood` feature: Any call to a selector for a class for which no method is defined results in a call to the `not-understood` method for this class. By default (i.e., inherited from `object`), `not-understood` just produces an error report (as demonstrated by `some-A bar`), but you can define your own method to deal with not-understood messages, and this is done here: The `not-understood` method for B just invokes the original selector (which is passed as xt) for `some-A1`, which eventually prints `bar-A1`.

Objects2 has a bunch of other features (e.g., for defining instance variables), but they do not play a role for the issues discussed in this paper, so they are not discussed here.

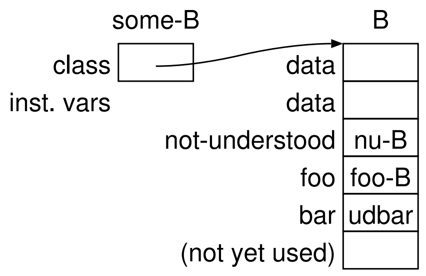


Figure 2: Object `some-B` and it's class `B`. `Some-B` has no instance variables

3 Dispatch Techniques

3.1 Unhashed general selectors

If any selector can be called with an object of any class, we have to implement a class×selector matrix. Figure 1 also shows the matrix for the example program. Some entries are defined directly by the programmer (e.g., the entry for `foo`×`A`), some are defined by inheritance (e.g., `foo`×`A1`); the rest gets the xt of the word `ud`*`selector`* (short for `undefined-`*`selector`*) which calls `not-understood` for the class and passes it the xt of *`selector`*.

In practice, instead of creating one big matrix, we store each column in the data of its class (see Fig. 2). Each object starts with a pointer to this class data. So the code for dispatching an unhashed method is:

```
: unhashed-selector ( u-offset "name" -- )
  create ,
does> ( ... object -- ... )
  ( object selector-body )
  @ over @ + ( object xtp ) @ execute ;
```

We cannot resize the class after objects of the class have been created: resizing might require moving the class data, i.e., updating the class pointers in the objects; since we do not track objects, we cannot do that. Therefore, we specify in advance how many unhashed selectors there are (see Section 6).

VFX Forth translates a call to such a selector into:

```
MOV  EDX, 0 [EBX]
ADD  EDX, [<selector-body>]
CALL 0 [EDX]
```

The memory access to the selector body cannot be optimized away by VFX, because the user is allowed to change the offset there at any time. However, it is possible to define selectors in a way that avoids that problem:

	Classes	Selectors	Cells
Minos	129	364	46956
GlForth	29	79	2291

Figure 3: Memory consumption of unhashed general selectors

```
: do-unhashed-selector ( object offset -- )
 over @ + ( object xtp ) @ execute ;

: unhashed-selector ( u-offset "name" -- )
 >r : r> postpone literal
 postpone do-unhashed-selector postpone ; ;
```

VFX compiles a call to such a selector into

```
MOV  EDX, 0 [EBX]
CALL [EDX+<u-offset>]
```

Unfortunately, this version is only fast on compilers that inline calls and optimize the result, like VFX.

Figure 3 shows the memory consumption of the dispatch tables of unhashed general selectors. For large programs the size of the dispatch tables can become a problem, because it grows approximately quadratically with the size of the program.

3.2 Hashed general selectors

Larger programs have more classes and more selectors, and usually the matrix is sparsely populated, i.e., most matrix entries point to ud*selector*. To save memory, we can use a hash table for looking up all the entries that are not ud*selector*; if no entry is found in the hash table, we call `undef` (a generic variant of ud*selector*), which eventually calls `not-understood`. As a key into the hash table, we can use an integer computed from a class index and a selector index: the class indices are spread so far apart that a class index can be just added to the selector index to get a unique key. Fig. 4 shows a hash table for our example.

The code for the hashed dispatch is:

```
does> ( ... object -- ... )
 @ ( ... object sel-id )
 over object-class @ class-base @ +
 ( object key )
 tuck hash-multiplier um* +
 ( key object hash )
 table-mask and 2* cells meth-hash-table +
 rot begin ( object table-entry key )
  over @ over = if \ right class/selector?
   drop cell+ @ execute exit then
  over @ 0= if
   nip undef exit then
  swap cell+ cell+ swap
 again ;
```

key	value
B::foo	foo-B
A1::foo	foo-A
B::not-understood	nu-B
A::foo	foo-A
A::not-understood	nu-obj
object::not-understood	nu-obj
A1::not-understood	nu-obj
A1::bar	bar-A1

Figure 4: A hash table for our example program

First this computes the key, then this key is hashed with a simple hash function, then we perform linear probing in the hash table, until the key matches our class/selector pair (then we `execute` the method), or until we find an empty entry (then we call `undef`). Note that the search loop is typically iterated very few times (ideally 0 times).

Whether the hashed or the unhashed version is preferred depends on the memory and run-time requirements of the application. E.g., if we assume that each selector in Minos has four methods on average, and that these methods are inherited to four classes on average, then we have 5824 entries in our hash table. We need either an 8K entry (16K cell) hash table with a 71% load factor (which may be slow), or a 16K entry (32K cell) hash table with a 36% load factor, but that does not save much memory compared to unhashed selectors.

Objects2 gives you the option of using the unhashed access for the most frequently used selectors (also useful for selectors that have methods for most classes), and hashing for the rest; see Section 6. By using unhashed selectors for the most frequent selectors, the relatively high load factor of the smaller hash table becomes acceptable, because only infrequent selectors are hashed; also, there are now fewer entries in the hash table, so the load factor is reduced somewhat.

```
class-selector u
class-selector v
class-selector w
class-selector x
object class
  :: u ." A-u" ;
end-class A
object class
  :: v ." B-v" ;
end-class B
A class
  :: w ." A1-w" ;
end-class A1
A class
  :: x ." A2-x" ;
  :: u ." A2-u" ;
end-class A2
```

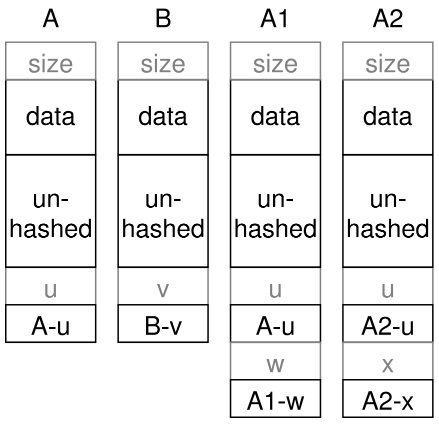


Figure 5: Class selectors. Different selectors (u and v, w and x) have the same index (optional checking data in gray)

3.3 Class selectors

Consider the following restriction: A selector can only be used on a specific class and its descendents. This means that two selectors for two non-overlapping classes (i.e., where neither class is descended from the other) can use the same index, resulting in densely populated dispatch tables (see Fig. 5) and lower memory consumption. We call these selectors *class selectors*.

We define classes starting with the most ancestral ones, and define all class selectors before we define child classes; this allows a very simple management of the selector indices: Every class has a current maximum selector index; defining a new class selector increases the maximum, thus creating an index for the new class selector. A child class inherits the maximum from its parent (and the parent's maximum stays the same from then on).

The dispatch code for class selectors without checking is the same as for unhashed selectors. The difference is in the index management and in the resulting restrictions: We have a limited number of unhashed selectors (the number is specified when loading objects2, see Section 6), whereas the class selectors are unlimited, but must satisfy the class selector restriction. In objects2 the indices of the class selectors start right after the indices of the unhashed selectors.

Using a class selector on an object of the wrong class will call the wrong method, or whatever is found at the class selector's offset from the start of the class; if we are lucky, we get a crash right away, if we are unlucky, the program does something we don't want.

We can have class selectors that check whether they are invoked for the right class: In addition to the method xt, we store the body address of the selector and the size of the class structure (shown in gray in Fig. 5). The selector then checks that its offset is within the class, and that what is stored right before the method is its body address. If not, the selector can produce an error (useful if checking is turned off after debugging) or perform not-understood processing. The selector code for the latter case is:

```
does> ( ... object -- ... )
 ( object sel-body )
 dup last-class-selector !
 tuck @ over object-class @
 ( sel-body object offset class )
 2dup class-size @ u< if
  ( sel-body object offset class )
  + rot over @ = if ( object p )
   cell+ @ execute exit
  then
  drop
 else
  2drop nip
 then ( object )
 last-class-selector @ cell+ @
 message-not-understood1 ;
```

With the parameters above (364 selectors, each with 4 methods that are inherited to 4 classes on average), class selectors consume 5824 cells of dispatch tables without checking and 11648 cells with checking. However, to work around the class selector restriction, programmers are likely to create deeper inheritance hierarchies and define selectors higher in the hierarchy, so the memory savings of class selectors are probably less than would be expected from the simplistic calculation above. The extreme variant of this would be to have a common ancestor class for all classes and define all selectors there, so all selectors are available for all classes, but with the same memory consumption as unhashed general selectors (for the unchecked version; checking is not necessary in this case).

```
interface
  selector i1
  selector i2
end-interface I
interface
  selector j1
  selector j2
end-interface J

object class
  implements I
  :: i2 ." A-i2" ;
end-class A
object class
  implements J
  :: j1 ." B-j1" ;
end-class B
B class
  implements I
  :: i1 ." B1-i1" ;
end-class B1
```

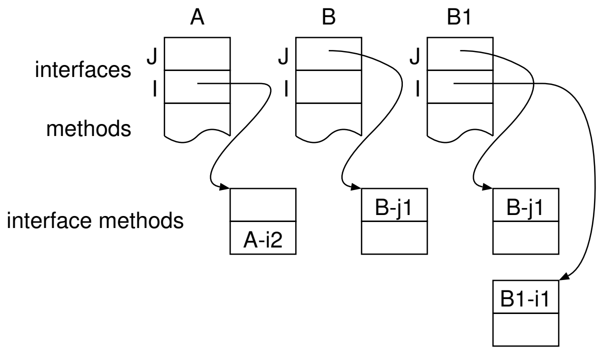


Figure 6: Interfaces and their implementation

3.4 Interface selectors

The original `objects.fs` did not have general selectors, so it included interfaces to make it possible to go beyond the limitations of class selectors: An interface is a set of selectors; you can define any class to support an interface, and the selectors of the interface can then be called for the class and its descendents. Figure 6 shows an example. The code for (unchecked) interface dispatch is:

```
does> ( ... obj -- ... )
 ( obj sel-body )
 2dup @ ( obj sel-body object if-offset )
 swap @ + @
 ( obj sel-body if-table )
 swap cell+ @ + @ execute ;
```

Here the selector stores (first cell) the offset of the interface from the class pointer, and (second cell) the offset of the method from the interface pointer, and uses both offsets to access the xt of the method.

Every interface requires a cell in every class (not just those that implement the interface); there can be several selectors per interface, so interfaces are somewhere between class selectors and unhashed general selectors in functionality and memory consumption.

Objects2 has general selectors, and interfaces do not appear to add enough to justify the additional complexity, so objects2 emulates interfaces with general selectors (for compatibility with `objects.fs`).

3.5 Monomorphic selectors

Sometimes a programmer defines a method (and, implicitly, a selector) to keep the program flexible, but does not define another method for the selector for now. Then the selector is actually used monomorphically, and dispatch can be very simple:

```
does> ( ... object -- ... )
 @ execute ;
```

In other words, a monomorphic selector is a deferred word (this version does not check that only descendents of the class for which the method was defined are passed to the selector).

4 Implementing the current object pointer

Apart from method dispatch, there is another interesting implementation issue:

Like `objects.fs`, objects2 has a current object `this`. `This` is set on method entry from the top of stack, and is visible inside the method.

This sounds like an ideal use for locals (in particular, `(local)`), but there is one catch: Standard programs must not use more than one locals definition per colon definition. And unfortunately there are Forth systems like VFX that rely on and enforce this restriction. So if we use locals for `this`, the programmer cannot use locals inside methods.

The other alternative is to define `this` as a value. The disadvantages are that this approach requires hardening against exceptions which may be difficult in some cases, and multi-tasking would require a user value (or user variable), which may be slower than global values.

Another property of the value implementation is that it allows us to access instance variables from outside methods; this has benefits in debugging, and can also be used to access instance variables from ordinary colon definitions, which can be used for factoring or for converting non-object-oriented code to object-oriented code. This kind of usage also has dangers, and some may prefer a local `this` because it prevents this usage.

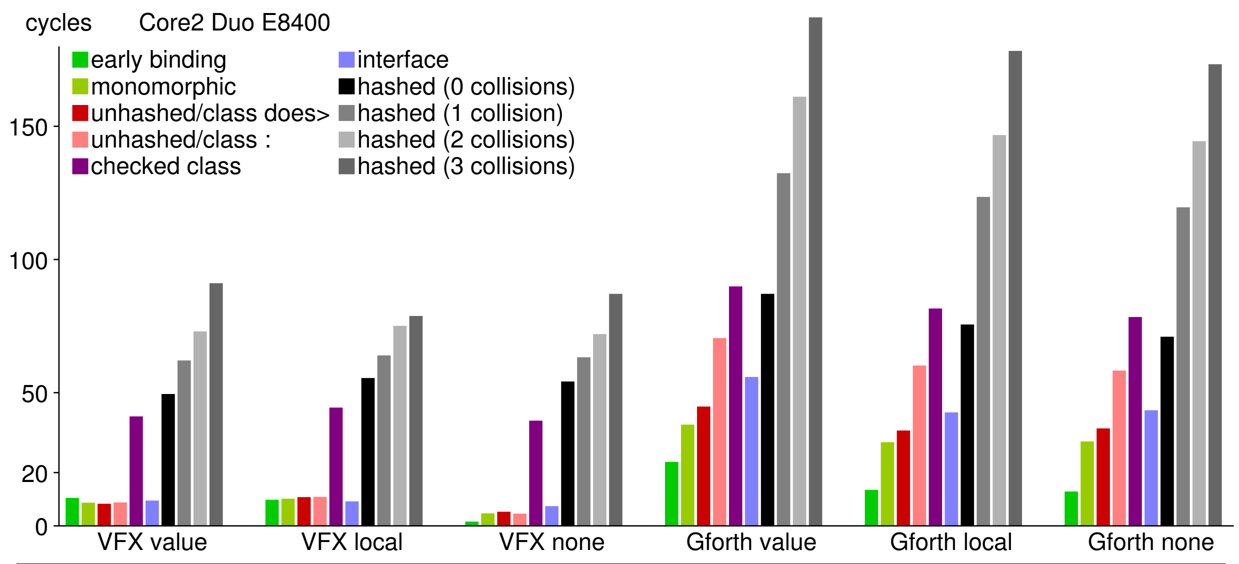


Figure 7: Time for one method call (plus overhead) in the micro-benchmark, varying dispatch code, Forth system, and `this` implementation

5 Execution time results

To compare the run time of various method dispatch (and current object pointer) techniques, I wrote a microbenchmark. It's a simple loop whose body calls the same selector 10 times in a row, and the object for which it is called is always the same (the method pushes that on the stack). I.e., caches should be hit and indirect branch prediction should be optimal.[2] The called method just increments the top-of-stack and pushes an object on the stack, but of course it does the handling of `this` (except for the *none* variant, which just `drop`s the object from the stack and pushes the object again). The shown times include the loop overhead around the selector calls. We also compare with *early binding*, where the method is compiled directly into the loop (and VFX inlines it) instead of going through some selector code. For the hashed selector, four timing variants are measured (by initializing the hash table appropriately): with 0, 1, 2 and 3 collisions (for this particular lookup) when probing the hash table; for a load factor of 50% (the value I recommend), most lookups should have zero or one collision.

Figure 7 shows the times in cycles per iteration, on a 3GHz Core2 Duo E8400. Two different Forth systems are used: VFX, an analytic native-code compiler that produces fast code for straight-line code; and Gforth, a system with a simpler code generation strategy (concatenate C-compiler-generated code fragments). Also, the two variants of implementing `this` are compared with each other, and for perspective, we also compare with not having `this` and (in this case) just dropping the object passed into the method.

All dispatch techniques except checked class selectors and hashed selectors have about the same performance on VFX, except that early binding is particularly fast for the *none* case, because VFX manages to optimize most of the loop body away: it inlines all the calls, and then VFX optimizes nearly all the dropping and pushing of the object away, leaving just the increments.

Interestingly, even though the VFX code for the unhashed selector using `:` looks much better than when using `does>`, this is not reflected consistently in the timing data.

Looking at the other dispatch techniques, on VFX a value `this` costs about 3–4 cycles more than no `this` and a local `this` costs about two more cycles. With more substantial methods, an out-of-order CPU like the Core 2 Duo will probably overlap the `this`-handling overhead with other code, reducing the cost for `this` even further.

The checked class selelector and the hashed selector are quite a bit slower on VFX: 30–35 cycles slower for the checked class selector, 43–50 cycles slower for the hashed selector with zero collisions. Each collision adds 9–10 cycles on average. The difference from the other selectors is surprisingly large, especially given the high speed of the other selectors. I believe this is mainly due to the fact that VFX's register allocation is limited to straight-line code (the other selectors all perform straight-line code). Hardware optimizations in the CPU might also play a role, even though the benchmark was modified so that the loop stream detector [Int12]

[2]That's not realistic, but indirect branch prediction should affect every technique in the same way and cache misses should be relatively rare for frequently-executed code; and rarely executed code does not have a significant influence on performance.

```
\ 3 unhashed selectors
   3 constant objects2-unhashed-selectors
\ hash table size: 2048
  11 constant objects2-hash-table-shift
\ warn if >1200 methods in hash table
1200 constant objects2-max-occupation

require objects2.fs

\ declare three selectors, such that they
\   are unhashed
selector draw
selector foo
selector bar
\ declare class and monomorphic selectors
class-selector baz
monomorphic-selector flip

\ load class libraries
require graphical.fs \ graphical class
require wine.fs      \ class about wine

\ load application code
require bla.fs
require blubb.fs
```

Figure 8: Choosing the implementation of selectors

should not come into play.

Unlike VFX, Gforth shows differences in performance between early binding, monomorphic, unhashed, and interface selectors, with the unhashed selector (implemented with `does>`) being 21–24 cycles slower than early binding. Class selectors using `:` as shown are significantly slower, because Gforth does not inline. The extra cost for checked class selectors is 40–45 cycles, and hashed dispatch without collisions costs 32-40 cycles more than unhashed dispatch, and each collision adds 34 cycles on average.

In Gforth a local `this` is faster than a value `this` by about 10 cycles, and not dealing with this is another 0–12 cycles faster.

Comparing unchecked class and interface selectors (from `objects.fs`) with unhashed and hashed selectors (new in objects2), we see that the added flexibility of the objects2 selectors either costs space (for the unhashed selector) or time and not as much space (hashed selector). Whether these costs are acceptable and whether the flexibility is worth the cost depends on the application and its environment. One of the features of objects2 is that it is easy to switch between these different selector variants, on a per-selector basis, as discussed in the following section.

6 Optimizing Dispatch

Objects2 offers the choice of using unhashed or hashed general selectors, class selectors, or monomorphic selectors. Moreover, you can make the choice on a per-selector basis, in a minimally invasive way: You do not need to change the class or method definitions, which may be libraries which you may not want to change. Instead, you can specify in the load file of the application which selectors use which dispatch implementation. The rest of this section describes this feature.

By default selectors are general selectors. The first n selectors are unhashed, the rest is hashed. You can determine the unhashed selectors by setting n before loading `objects2.fs` and then declaring the n selectors that you want to be unhashed.

You can also set the number of classes and the hash table size to reduce the memory consumption to the necessary amount or to allow more than the default number of classes and hash table entries.

You can also declare a selector as a class selector or monomorphic selector in the load file.

Here is an example of how a load file might look:

Note that these selector declarations happen outside any class, they just influence what the later method definitions do. Actually, implementation-wise, these “declarations” define a selector word of the desired kind, and a later method definition essentially just defines the method and inserts the xt into the appropriate table for this selector and class; in case of a class selector the first method definition inside a class also sets the root class for this selector, and every other method definition has to be in a descendent class of that class.

7 Missing language features

There are two language features that would be useful for implementing objects2 and which Forth systems usually provide in some way, but which are not standardized:

`body> ( addr -- xt )` would allow getting the xt of the selector for not-understood processing. But since this is not standardized, every selector has to store its xt in an extra field. Absence cost: one extra cell per selector.

`>definer ( xt -- definer )` would allow to check if a word has been defined as a selector or not, and which kind of selector. But since this word is not standardized, objects2 maintains several linked lists, one for each kind of selector, and if it needs to know the information, it searches these linked lists. Absence cost: another extra cell per selector, more CPU consumed by linear searches.

8 Related work

There is a large body of work on implementing object-oriented languages in general and method dispatch in particular [ACFG01, DH96, VH96, AGS94, ZCC97]. Ducournau [Duc11] presents a very good survey, but it helps to be familiar with some of the implementation techniques in order to understand this survey. The present work does not introduce any new techniques; instead, it makes a few of the existing techniques available to Forth programmers in a way that allows them to switch between different techniques easily, as appropriate for the application.

There have also been a number of object-oriented Forth extensions. Rodriguez and Poehlman [RP96] list 23, and since then more have been introduced, including `objects.fs` [Ert97]. This paper focuses on the main feature where objects2 differs from `objects.fs`: Allowing selectors to be used on arbitrary classes; it discusses the implementation of this feature and presents performance data.

A relatively recent entry in the collection of object-oriented Forth extensions is FMS (Forth meets Smalltalk). Like objects2, it supports defining methods for a given selector for any class. The implementation is not much documented, but seems to be based on a compressed table. I do not understand the table format enough to evaluate its space consumption. The dispatch code is relatively long compared to the variants shown above, so I expect it to take at least as much time as objects2's hashed dispatch with 0 collisions. A more substantial comparison is future work.

9 Conclusion

Fast, small, flexible (duck typing): Pick any two.

Fast, small: Class selectors, monomorphic selectors

Fast, flexible: Unhashed general selectors

Small, flexible: Hashed general selectors

Objects2 allows you to choose between these method selector implementations. Moreover, you can choose separately for each selector, and you can change the choice easily, in many cases not even touching the actual class code.

References

[ACFG01] Bowen Alpern, Anthony Cocchi, Stephen Fink, and David Grove. Efficient implementation of java interfaces: Invokeinterface considered harmless. In *Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '01)*, pages 108–124, 2001.

[AGS94] E. Amiel, O. Gruber, and E. Simon. Optimizing multi-method dispatch using compressed dispatch tables. *ACM SIGPLAN Notices*, 29(10):244–244, October 1994.

[DH96] Karel Driesen and Urs Hölzle. The direct cost of virtual function calls in C++. In *Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '96)*, pages 306–323, 1996.

[Duc11] Roland Ducournau. Implementing statically typed object-oriented programming languages. *ACM Computing Surveys*, 43(3):Article 18, April 2011.

[Ert97] M. Anton Ertl. Yet another Forth objects package. *Forth Dimensions*, 19(2):37–43, 1997.

[Int12] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, April 2012. Order number 248966-026.

[RP96] Bradford J. Rodriguez and W. F. S. Poehlman. A survey of object-oriented Forths. *SIGPLAN Notices*, pages 39–42, April 1996.

[VH96] Jan Vitek and R. Nigel Horspool. Compact dispatch tables for dynamically typed object oriented languages. In Tibor Gyimóthy, editor, *Compiler Construction (CC'96)*, pages 309–325, Linköping, 1996. Springer LNCS 1060.

[ZCC97] Olivier Zendra, Dominique Colnet, and Suzanne Collin. Efficient dynamic dispatch without virtual function tables. the SmallEiffel compiler. In *Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '97)*, pages 125–141, 1997.

net2o: Transport Layer — Implemented

Tame the Net

Bernd Paysan

EuroForth 2012, Oxford

Overview

1. Motivation
2. Datenflusssteuerung
3. Reliability
4. Cryptography

Recap: What's Broken?

- TCP–Flow Control: "Buffer Bloat"
- TCP as "carefree protocol" is not even remotely real–time capable, so far from "carefree" for media use
- UDP is only a "easy" access to raw IP, and otherwise "do it yourself"
- The SSL–PKI with their "honest Achmeds" as certification authorities
- Encryption "too complicated, too difficult", usually added late, and therefore way too often not done

Changes from the Draft

- Packet size now $64 * 2^n$, $n \in \{0, \ldots, 15\}$, so up to 2MB in powers of 2
- No "embedded" variant implemented, only 64 bit addresses
- Routing address length changed to 128 bits
- Encryption always active
- No "salt" at the start of a packet, but a cryptographic checksum (128 bit) at the end

Status: TCP Flow Control

- TCP fills the buffer, until a packet has to be dropped, instead of reducing rate before. Name of the symptom: "Buffer bloat". But buffering is essential for good network performance.

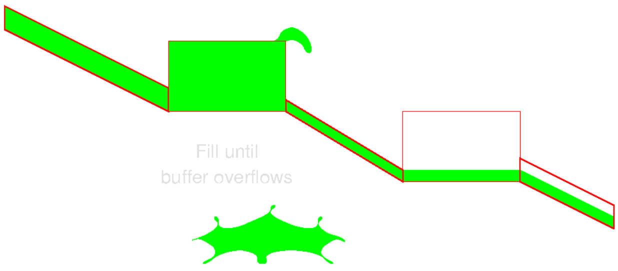


Figure: Buffer Bloat

Alternatives?

- LEDBAT tries to achieve a low, constant delay: Works, but not good on fairness
- CurveCP has a similar approach, which is not even documented (but Dan Bernstein's code is by definition "obvious")
- Therefore, something new has to be done

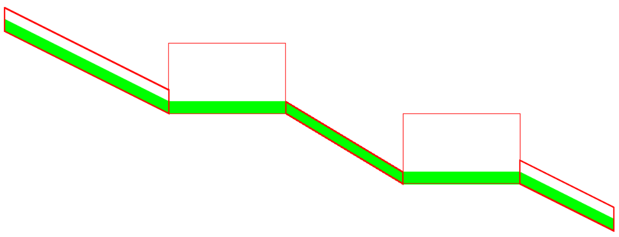


Figure: That's how proper flow control should look like

„Buffer Bloat"

- Retransmits are making the situation worse in case of congestions and therefore should be avoided
- Riddle: How big should the buffer be, under the assumption that the bandwidth is used optimally,the bottleneck is on the other side of the connection, and a second data stream is opened up?
- Answer: about half the round trip delay, which are inevitably filled before any reaction is possible
- Buffers are good, but you shouldn't fill them up to the brim
- The problem is inherent in the TCP protocol, but since Windows XP did not provide window scaling, the per–connection buffer limit was 64k for most connections on the Internet for quite a long time.

net2o Flow Control

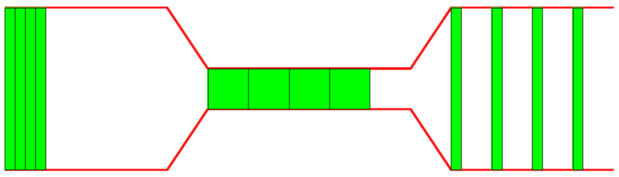


Figure: Measure the bottleneck using a burst of packets

Client Measures, Server Sets Rate

Client recores the *time* of the first and last packet in a burst, and calculates the achieved rate for received packets, extrapolating to the achievable rate including the dropped packets. This results in the requested *rate*.

```
: calc-rate ( -- )
  delta-ticks @ tick-init 1+ acks @ */
  lit, set-rate ;
```

Server would simply use this rate

```
: set-rate ( rate -- )  ns/burst ! ;
```

Fairness

Fairness means that concurrent connections achieve about the same data rate, sharing the same line in a fair way.

- Ideally, a router/switch would schedule buffered packets round–robin, giving each connection a fair share of the bandwidth. That would change the calculated rate appropriately, and also be a big relieve for current TCP buffer bloat symptoms, as each connection would have its private buffer to fill up.
- Unfortunately, routers use a single FIFO policy for all connections
- Finding a sufficiently stable algorithm to provide fairness
- We want to adopt to new situations as fast as possible, there's no point in anything slow. Especially on wireless connections, achievable rate changes are not only related to traffic.

net2o Flow Control — Fair Router

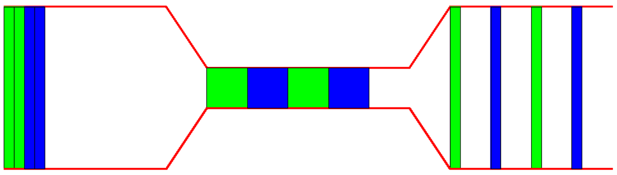


Figure: Fair queuing results in correct measurement of available bandwidth

net2o Flow Control — FIFO Router

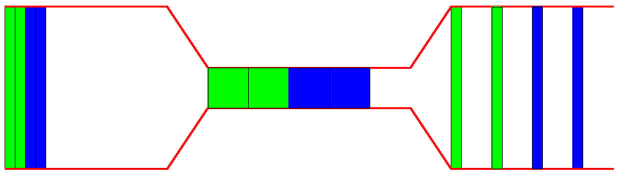


Figure: Unfair FIFO queuing results in twice the available bandwidth calculated

Fairness I

- To improve stability of unfair queued packets, we need to improve that P regulator (proportional to measured rate) to a full PID regulator
- The integral part is the accumulated slack (in the buffer), which we want to keep low, and the D part is growing/reducing this slack from one measurement to the next
- We use both parts to decrease the sending rate, and thereby achieve better fairness
- The I part is used to exponentially lengthen the rate Δt with increasing slack up to a maximum factor of 16.

$$s_{exp} = 2^{\frac{slack}{T}} \quad \text{where } T = \max(10ms, \max(slacks))$$

Fairness D

- To measure the differential term, we measure how much the slack grows (a Δt value) from the first to the last burst we do for one measurement cycle (4 bursts by default, first packet to first packet of each burst)
- This is multiplied by the total packets in flight (head of the sender queue vs. acknowledged packet), divided by the packets within the measured interval
- A low–pass filter is applied to the obtained D to prevent from speeding up too fast, with one round trip delay as time constant
- $\max(slacks)/10ms$ is used to determine how aggressive this algorithm is
- Add the obtained Δt both to the rate's Δt for one burst sequence and wait that time before starting the next burst sequence.

VDSL

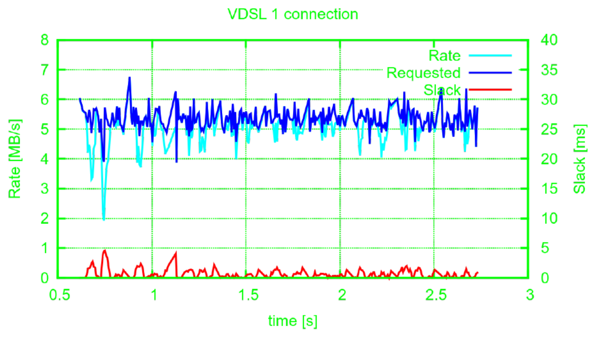


Figure: One connection on a VDSL

VDSL, Congestion

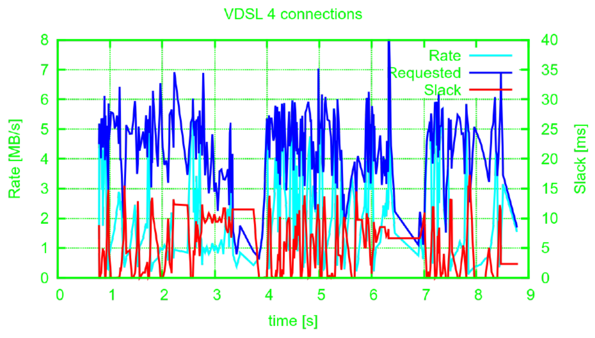


Figure: One of four connections on a VDSL

Unreliable Air Cable (WLAN)

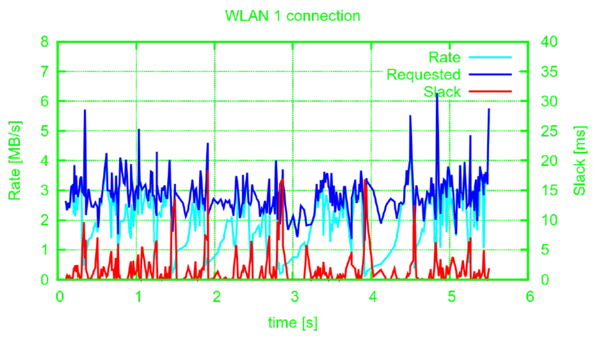


Figure: Single connection using WLAN

Unreliable Air Cable, Congestion

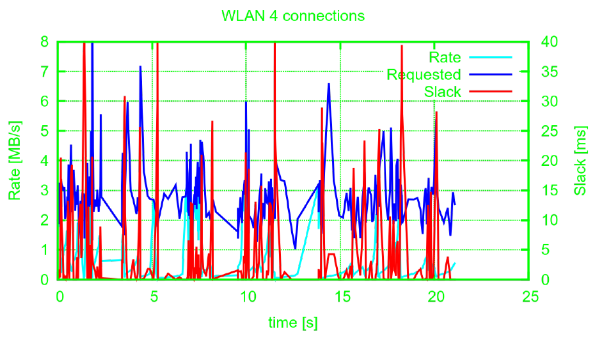


Figure: One of four connections using WLAN

Transport Reliability

- Packet ordering is dealt with the address each packet carries
- The receiver tracks received packets in two alternating bitmaps
- Received packets are marked as received in the active bitmap
- The other bitmap is filled up, until the bitmaps are swapped (twice per round trip delay RTD)
- Wait one RTD for retransmits
- Retransmits are preferred, but no timing measurement on retransmits (two identical packets in flight)

Reliable Execution of Commands

- The command block at that address is received first time $\longrightarrow$ execute, remember the reply command
- The command block has already been received $\longrightarrow$ send the reply again (don't execute the command)
- No replies requested $\longrightarrow$ Do nothing
- Acknowledges are amended by a checksum, which only the sender or the receiver can compute, so no fake acknowledge for dropped packets is possible.

Cryptography

Communication needs the first three goals, the fourth one isn't

Confidentiality no third party (Eve) should eavesdrop the communication

Integrity The data is complete and unmodified

Authentication The sender of the data can be identified

Non–repudiation is not necessary for two–way communication

Used Technology: Curve25519

- Elliptic Curve Cryptography doesn't base on large number factoring (as hard to solve problems), but on natural logarithms of elliptic curves
- Security level of Curve25519 corresponds to 128 bits in a symmetric key — that's sufficient today
- Curve25519 has a very efficient implementation
- It is optimized for 1:1 connections
- Each participant "multiplies" his secret key with the public key of the other side, both products are identical

Wurstkessel

At the moment, I'm using Wurstkessel as symmetric encryption, even though there hasn't been a thorough review:

- Wurstkessel provides en/decryption and authentication in a single pass, computing a key–dependent secure hash
- Thus a single run of Wurstkessel solves all three tasks: confidentiality, integrity, and authentication.
  1. the data is encrypted
  2. the correct hash proves its integrity
  3. the hash can only be calculated knowing the key, therefore proving the authentication of the sender
- AES has something similar, the CBC–MAC. However, in AES, it is necessary to use different keys for encryption and MAC, i.e. no single run possible

Hidden Initialization Vectors

- No key reuse allowed (only for retransmissions), otherwise a known–plaintext attack is possible
- Usual approach: initialization vector (IV) transmitted with each packet
- Disadvantage: Overhead and the "other" part of the key is known to the attacker
- Solution: Generate the IVs using a PRNG (with Wurstkessel in PRNG mode) on both sides — these IVs are "shared secrets". Only the seed for the PRNG is transmitted, and used together with the shared key to generate the IVs (Idea: Helmar Wodke).

Public Key Infrastructure (PKI)

At the moment, three approaches are used:

1. Hierarchical Certification Authorities (e.g. SSL): The trust is delegated to “notaries”, i.e. the CAs, which then must be trustworthy (all of them, since each CA can create a certificate for anybody). The server is certified, i.e. the user knows that he can trust this connection as much as the worst of those 600 CAs.
2. Peer to Peer (e.g. PGP): trust is obtained through a “web of trust”, i.e. you either trust directly or by using several people you trust. It is not sufficient to corrupt a single person in your trust network to obtain trust.
3. Observing changes (e.g. SSH): trust is reiterated by repeated contacts, and as long as keys don’t change, trust is assumed.

What Was the Problem?

The typical reason to use a trusted connection is to obtain a secure login, and then access private data. This begs a question:

- Isn’t it actually the *client*, which should be trusted?

The connection is a trusted connection, if *one* participant has successfully evaluated the trust of the other.
Therefore, by inverting the trust relation, the SSH approach is sufficient in most cases.

For Further Reading

Bernd Paysan
Fossil Repository und Wiki
http://fossil.net2o.de/

Recognizers
Customize the Interpreter

Bernd Paysan

EuroForth 2012, Oxford

Overview

The Problem

- Forth is extensible, provided all your extensions are simple, space–delimited words
- Literals are already part of the non–extensible, unchangeable part of the standard interpreter
- Many systems have mechanisms like `notfound`, where you can plug in something in a system–dependent way...

Recent Development

- During the number prefix RfD discussion, Anton Ertl [1] suggested a system called "Recognizer," which was roughly sketched, but would allow to dynamically reconfigure the interpreter
- Matthias Trute had several discussions on IRC and implemented a recognizer system in amForth[1]
- Win32Forth got recognizers in the current development snapshot, as well as Gforth
- All these recognizers look slightly different, as they are still experimental stuff

Gforth's Recognizers

x–RECOGNIZER (addr u | token r:x / addr u r:fail)
: A recognizer takes a string, and converts it to a token, which consist of some data on the stack and a method table. The method table have three "virtual" methods (which are only concept):

INT (x*i token — y*j)
: Invokes the interpretation semantics of a token (similar to EXECUTE)

COMP (token —)
: Invokes the compilation semantics of a token

LIT (token —)
: Add the token to the currently defined word, so that tokens can be postponed

Gforth's Recognizers

RECOGNIZER: (xt-int xt-comp xt-lit „name" —)
: Creates a recognizer table

Recognizers are organized as a stack (similar to wordlists), therefore you can

GET–RECOGNIZERS (rec–addr — rec$_n$ .. rec$_1$ n)
: get the all the recognizers out of a stack

SET–RECOGNIZER (rec$_n$ .. rec$_1$ n rec–addr —)
: set the recognizers of a stack

Gforth's Recognizers

DO–RECOGNIZER (addr u rec–addr — token r:table | addr u r:fail)
: walks through all the recognizers in a stack until one matches, and either return its result or the input string and r:fail

R:FAIL (– r:fail)
: recognizer table, where all three methods fail with `-13 throw`

Predefined Recognizers: Forth words

```
: lit, ( n -- ) postpone Literal ;
: nt, ( nt -- ) name>comp execute ;
: nt-ex ( nt -- ) name>int execute ;
' nt-ex ' nt, ' lit, recognizer: r:word
: word-recognizer ( addr u -- nt r:word | addr u r:fail
  2dup find-name
  [ [IFDEF] prelude-mask ] run-prelude [ [THEN] ] dup
  IF  nip nip r:word  ELSE  drop r:fail  THEN ;
```

Predefined Recognizers: Literals

```
: 2lit, postpone 2Literal ;
' noop ' lit, dup recognizer: r:num
' noop ' 2lit, dup recognizer: r:2num
: num-recognizer ( addr u -- n/d table | addr u r:fail )
  2dup 2>r snumber?  dup
  IF  2rdrop 0> IF r:2num ELSE r:num THEN EXIT  THEN
  drop 2r> r:fail ;
```

Advanced Recognizers: Strings

```
: slit,  postpone sliteral ;
' noop ' slit, dup recognizer: r:string
: string-recognizer
    ( addr u -- addr' u' r:string | addr u r:fail )
    2dup s\" \"" string-prefix?
    IF    drop source drop - 1+ >in !
          \"-parse save-mem r:string
    ELSE  r:fail  THEN ;
' string-recognizer
forth-recognizer get-recognizers
1+ forth-recognizer set-recognizers
```

For Further Reading

Anton Ertl
Usenet Posting *number parsing hooks*
https://groups.google.com/forum/?fromgroups#!msg/comp.lang.forth/r7Vp3w1xNus/Wre1BaKeCvcJ

Matthias Trute
Recognizer — Interpreter dynamisch verändern
VD 2011/02

Bernd Paysan
Recognizer
VD 2012/02