24th EuroForth Conference

September 26-28, 2008

TU Wien
Vienna, Austria

Preface

EuroForth is an annual conference on the Forth programming language, stack machines, and related topics, and has been held since 1985. The 24th EuroForth finds us in Vienna for the second time. The three previous EuroForths were held in Santander, Spain (2005), in Cambridge, England (2006), and in Schloss Dagstuhl, Germany (2007). Information on earlier conferences can be found at the EuroForth home page (`http://dec.bournemouth.ac.uk/forth/euro/index.html`).

Since 1994, EuroForth has a refereed and a non-refereed track.

For the refereed track, one paper was submitted, and one was accepted (100% acceptance rate). For more meaningful statistics, I include the numbers from 2006 and 2007: eight submissions, four accepts, 50% acceptance rate. The paper was sent to three program committee members for review, and they produced three reviews. This year, one of the program committee members has submitted a paper; of course this member was not involved in the review process of the paper in any way. I thank the authors for their paper, and the reviewers for their reviews.

Several papers were submitted to the non-refereed track in time to be included in the printed proceedings. In addition, the printed proceedings include the slides and an abstract for talks that will be presented at the conference without being accompanied by a paper. These online proceedings also contain late papers and late presentations that were too late to be included in the printed proceedings. One presentation (GLforth by Gerald Wodni) was in demo format and is therefore not included in these proceedings. Workshops and social events complement the program.

We are grateful to Ewa Vesely for organizational support for this year's EuroForth.

Anton Ertl

Program committee

Sergey N. Baranov, Motorola ZAO, Russia (secondary chair)
M. Anton Ertl, TU Wien (chair)
David Gregg, University of Dublin, Trinity College
Ulrich Hoffmann, FH Wedel University of Applied Sciences
Jaanus Pöial, Estonian Information Technology College, Tallinn
Bradford Rodriguez, T-Recursive Technology
Bill Stoddart, University of Teesside
Reuben Thomas, Adsensus Ltd.

Contents

Refereed papers

Non-refereed papers

Abstracts

Presentations

Late papers

Late presentations

Programme

Thursday	1:00pm	Lunch at Artner
	2:30pm	Forth200x meeting
	7:00pm	Dinner at Wiazhaus
Friday	9:00am	Forth200x meeting
	1:00pm	Lunch at Artner
	2:30pm	Coffee at meeting room
	2:45pm	EuroForth opening, organizational matters
	3:00pm	Angel Robert Lynas, Bill Stoddart: Using Forth in a Concept-Oriented Computer Language Course
	3:30pm	Jaanus Pöial: Java Framework for Static Analysis of Forth Programs
	4:00pm	Andrew Haley: An Infix Syntax for Forth (and its compiler)
	4:30pm	Coffee break
	5:00pm	Bernd Paysan: A Debugger for the b16 CPU
	5:30pm	M. Anton Ertl: Cleaning up After Yourself
	6:00pm	Ulrich Hoffmann: Forth System Hooks—Metaobject Protocol in Forth Style
	7:00pm	Meet at hotel for walk to dinner
	7:30pm	Conference Dinner at Zwölfapostelkeller
Saturday	9:00am	Stephen Pelc: Updating the Forth Virtual Machine
	9:30am	Yunhe Shi, Kevin Casey, M. Anton Ertl, David Gregg: Virtual Machine Showdown: Stack versus Registers
	10:00am	Bernd Paysan: Porting MINOS to VFX
	10:30am	Coffee break
	11:00am	Bill Stoddart, Angel Robert Lynas: Filters for Unicode Markup, a Work in Progress Report
	11:30am	Gerald Wodni: GLforth – an Ego Shooter
	11:55am	Impromptu talks/discussion
	12:50pm	Walk to restaurant
	1:00pm	Lunch at Wieden Bräu
	2:30pm	Excursion to Leopoldsberg, Kahlenberg, Cobenzl, followed by Dinner at *Weingut am Reisenberg*
Sunday	9:00am	EuroForth workshops and impromptu talks
	1:00pm	Lunch at Wieden Bräu
	2:30pm	4th day: Walk: Alte Donau, Donauinsel and Prater

Using Forth in a Concept-Oriented Computer Language Course

Angel Robert Lynas and Bill Stoddart
University of Teesside

September 8, 2008

Abstract

We describe a way of teaching fundamentals of Language Systems (for second-year Computing students), without having to compromise the use of a simple grammar owing to hardware limitations which need no longer apply in this setting.We adopt a top-down approach, reading from right-to-left and splitting the input string on the rightmost operator appropriate to that level. The target platform is Reversible Virtual Machine (RVM) Forth, so a postfix translation is the aim. We introduce a basic arithmetic grammar and expand it during the course to allow unary minus, floating point and function application; this shows how type information can be generated in one pass and resolved in a second, via an internal intermediate code. Each version of the grammar has its productions mapped onto a system of equations which serve as the specification for the implementation functions.

1 Introduction

In teaching Language Systems for Computer Science, the conceptual simplicities can often be obscured by the compromises made to accommodate purely historical restrictions on software writing for far more limited computers than those available today. These compromises take the form, for instance, of extra complications in the simple grammars, and trying to do several jobs in one pass of the compiler.

Accepting that we can now directly implement, at least in a pedagogical setting, a more straightforward approach less hampered by traditional constraints, can lead to a clarification of the conceptual issues involved in top-down parsing.

We used a basic grammar for infix arithmetical expressions, and used the conversion to postfix as a convenient route for exploring grammar analysis. An expansion to include floating point numbers and other facilities proved to be a fruitful method for introducing the usage of type information and two-pass compiling with intermediate code to hide the use of meta-information.

Following some definitions, we describe in Section 2 the overall approach taken, then in Section 3 the basic grammar presented to the students. In Section 4 we examine the extensions to this grammar, the actions of the new compiler's first pass and those of its second. We conclude with a brief look at a possible future application of these ideas for subsequent refinements of the course.

1.1 Some Definitions

In the following, we use some fundamental terminology from the study of grammars:

Terminals These are the strings which are the tokens of the language and appear in generated expressions.

Non-Terminals These are symbols which do not appear in the output, but stand for classes of terminals, or combinations of those classes.

Productions Specifies one way in which a non-terminal can be expanded to a sequence of other non-terminals and/or terminals, eventually generating strings composed entirely of terminals.

2 Higher-Level approach

The usual approach in teaching about language systems has been to adopt a relentlessly left-to-right method, compatible with historical restraints on memory, storage and processing capacity. Thus in the interests of efficiency, perhaps one token is available for "look-ahead" while the current token is being processed (look-ahead by default referring to the token on the right). See for instance the popular compiler texts [1] and [2].

Using these techniques, everything is done in a single pass, including storage and checking of type information (if applicable) and resolution of conditional structure — though we don't consider the latter here.

Certain things must be sacrificed to this methodology, the first casualty being simplicity of grammar. A grammar for basic arithmetic could contain a production (we define <expression> and <term> more fully later)

<expression>::=<expression> + <term>

meaning that one way of constructing an expression is by combining an expression and a term with a "+" sign in between. This we generally abbreviate to

$$E ::= E + T$$

This, however, is not really suitable for left-right parsing owing to the left recursion; that is, an E could expand to $E + T + T + \cdots + T$, so reliably telling where the first <expression> ends is problematical. The order of E and T specifies left-associativity for the + operator and others, ensuring that (for instance) $a - b - c$ is parsed as $(a - b) - c$, rather than $a - (b - c)$ which gives a different result.

In order to remove this left-recursion, such constructs are usually redefined using dashed letters to denote “the rest of the expression”, now looking like this

$$E ::= TE'$$
$$E' ::= +TE' \mid \epsilon$$

The symbol ϵ or “*null*” stands for the empty string. The decomposition of the expression E can now be approached unambiguously from the left, as the term (processed by a similarly expanded rule) is recognised and the “rest of the expression” E' is passed downward for further processing. But this represents a considerable loss in simplicity compared to the first grammar.

What method, then, could be used to parse productions like $E \rightarrow E + T$ as is? Clearly a right to left approach would be more apropriate here, splitting the expression at the rightmost top-level “+” encountered, where the remaining expression on the left is dealt with by a recursive call to the expression parser, and the term is dealt with by the term parser. Terms are split on “*” or “/” if any occur, again the rightmost, for example $T = T * F$, where F is a factor with no top-level operations. We can now fill in the rest of the grammar and formalise the operation of a recursive parser.

3 A Simple Arithmetic Grammar

The initial grammar developed for the students, which allows for unsigned integers, identifiers, and bracketed expressions, is as follows. The order of the non-terminal expansions reflects the precedence order of the operators; lower precedence operators are scanned for first, and the non-terminal split if applicable. The higher precedence operators appear closest to the terminals in the postfix output and are thus executed first. The vertical bars on the left-hand side indicate alternative productions for the same non-terminal.

Non-Terminals:

E is an Expression; these can contain a plus or minus sign at the top level (i.e. not within any brackets).

T is a Term, which contains no pluses or minuses at its top level, but can contain “*” or “/” for multiply or divide.

F is a Factor, containing no top-level operations, but can expand to an unsigned number U, or an identifier I, or a bracketed expression (E), the contents of which are recursively expanded as an expression. We do not define U or I further at present; in any case U will be later replaced by a non-terminal which accommodates both integer and floating point numbers.

$$E ::= E + T \mid E - T \mid T$$
$$T ::= T * F \mid T/F \mid F$$
$$F ::= U \mid I \mid (E)$$

The compilation to postfix is described by a set of equations involving mutually recursive functions which act on appropriate strings of each non-terminal class. The right-hand sides of these equations show the output from a given string; at the top levels, this will involve invoking other functions on part of the string — possibly including the function itself recursively. In the following, lower-case e, t, f, u and i represent strings belonging to the nonterminals E, T, F, U and I respectively; that is, e is a particular expression, t is a particular term, and so on. We have the functions

P_E takes a string e and returns the translation of that string in postfix.

P_T takes a string t and returns the translation of that string in postfix.

P_F takes a string f and returns the translation of that string in postfix.

If no operators are found by P_E or P_T, they pass the entire string down to the next function. The symbol "$\frown$" is used for string concatenation. The mutually recursive equations which define the compilation are

$$P_E(e \frown \texttt{"+"} \frown t) = P_E(e) \frown P_T(t) \frown \texttt{" +"} \tag{3.1}$$

$$P_E(e \frown \texttt{"-"} \frown t) = P_E(e) \frown P_T(t) \frown \texttt{" -"}$$

$$P_E(t) = P_T(t)$$

$$P_T(t \frown \texttt{"*"} \frown f) = P_T(t) \frown P_F(f) \frown \texttt{" *"} \tag{3.2}$$

$$P_T(t \frown \texttt{"/"} \frown f) = P_T(t) \frown P_F(f) \frown \texttt{" /"}$$

$$P_T(f) = P_F(f)$$

$$P_F(u) = u$$

$$P_F(i) = i$$

$$P_F(\texttt{"("} \frown e \frown \texttt{")"}) = P_E(e)$$

At the top two levels — expression and term — the parsing is right to left; specifically, a Forth operation LSPLIT is used to search for a symbol in $\{+, -\}$ or $\{*, /\}$ at the top level. The stack parameters are the string to be searched and a sequence of strings which are the tokens to search for. An example call follows, but a brief explanation is needed first.

A string in RVM-Forth can be an "ASCII-Zero" or AZ string, which is terminated by an ASCII null (in other words, a 0 after the final character, like strings in C and some other languages). They are created with the $\boxed{\texttt{"}}$ word, terminated by another quote. Also, sequences are created with the syntax

```
<type> [ <element1> , <element2> , ]  ( and so on)
```

where the comma word allocates storage for each element. The type can be simple (integer, string) or composite, involving pairs and nested sets or other sequences.

An example call to LSPLIT using a string EXPR1, then, is

```
EXPR1 STRING [ " +" , " -" , ] LSPLIT
```

This would be an call to process EXPR1 using the symbols $\{+, -\}$.

Should LSPLIT encounter a right bracket ")", this means that a lower-level expression is included in the top-level one, which expression will be dealt with by a later recursive call to P_E. So LSPLIT will not continue its search for operators until it finds the matching left bracket; it will keep track of the level, incrementing for any right brackets and decrementing for matching left ones, until its own level zero is reached again.

On encountering a symbol/token in its search set, the operation splits the input string into a left part, the symbol string, and a right part. As can be seen from equations (3.1) and (3.2) above, the first part is sent recursively to P_E or P_T as appropriate, the second part to P_T or P_F, while the operation token is output last of all, so that it occurs *after* the outputs from parsing the rest of the string. If no search symbol is found, the entire string is passed down to the next parsing level.

When the factor level is reached, subsequent processing of U and I can be done left to right, as no left recursion occurs in these definitions. A factor of the form "(E)" merely has its brackets stripped and the contents sent back to the top-level function P_E. We include the code for P_T (called PT), which parses terms into terms and factors, as figure 1 below. The local variable syntax uses the words (: and :) to delineate the declarations, with values being taken from the stack.

```
: PT ( az1 -- az2, parse a term, leaving az2 the postfix
       translation of the term az1 )
   (: VALUE e :)
    e STRING [ " *" , " /" , ] LSPLIT
    VALUE BEFORE VALUE AFTER VALUE OP-STRING
    OP-STRING NULL =
    IF
      BEFORE PF      ( No op found, so e was a factor )
    ELSE
      BEFORE RECURSE  AFTER PF  ^  OP-STRING  ^
    THEN
  1LEAVE ;
```

Figure 1: RVM-Forth code for P_T.

The general technique is known as Recursive Descent Parsing, and is well-known, though we do not know of its having been implemented in this bidirectional way. The usual categories of LL(k) or LR(k) do not apply, strictly speaking, as they denote solely unidirectional parsing; for instance the Earley recogniser [3], based on Knuth's LR(k) algorithm. Our technique is adaptable insofar as right-associative operators can and will be accommodated by a sort of mirror image of LSPLIT which would read from left to right.

The parsing can be illustrated with a couple of examples, which can either be represented as parts of trees (useful for the students initially), or in a linear fashion which follows the equations closely. Given the input string "$x * x - 1 - (x - 1) * (x + 1)$", for instance, the first parsing step is shown in figure 2.

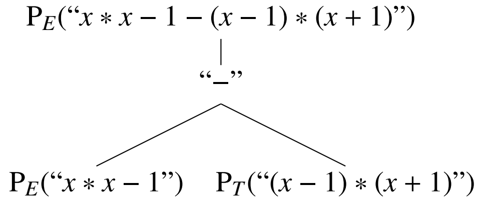


Figure 2: Splitting at the top level.

The left-hand side of this is then the input to a recursive call to P_E; the rest of this is shown (as far as the Factor level) in figure 3.

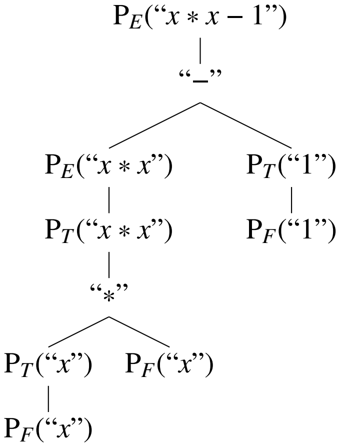


Figure 3: Splitting the left-hand half down to Factor level.

In linear style, the first part of this second tree would appear as

$$P_E("x * x - 1") = P_E("x * x") {}^\frown P_T("1") {}^\frown " - "$$

which we can see follows the pattern of equation 3.1.

As regards performance, we have not yet undertaken any exhaustive time complexity analyses. Some empirical run-throughs indicate that the basic technique is probably $O(n^2)$ for a suitable grammar; the limitations with respect to which grammars can be handled require further investigation, however.

4 Floating Point Extension

We now consider extending the grammar to include unary minus, function application and floating point capability. The inclusion of mixed-mode arithmetic requires the simple compiler to become a two-pass compiler, whereby the first pass generates intermediate code containing type information, as we now have integer and floating point to deal with. These require different operations for arithmetic and printing, and sometimes conversion is required from integer to floating point.

Unary minus is dealt with by converting the minus sign into a tilde "~", as the compiler uses this internally; it's converted back for the Forth-readable output, and allows multiple minuses in a row which are converted consistently. The details are not examined further here. Function application is of the form "<identifier>(<arg-list>)", with the arguments comma-separated.

The basic grammar is extended as follows, with the extra non-terminals

- F_0 This is simply an unsigned factor.
- N This replaces U, and is a general number (integer or float), parsed by the floating point state machine to be described later.
- L This is a comma-separated list of arguments to a function.

$$\begin{aligned}
E &::= E + T \mid E - T \mid T \\
T &::= T * F \mid T/F \mid F \\
F &::= F_0 \mid -F \\
F_0 &::= N \mid I \mid (E) \mid I(L) \\
L &::= L, E
\end{aligned}$$

The list of arguments L is parsed right to left, in a similar way to E and T, but split by `LSPLIT` on a comma.

4.1 First Pass (intermediate code)

The generating functions for the upper levels are similar for those in the previous section, but instead of including a straightforward plus or multiply sign in the output, they include the string for an internal operation which the second pass of the compiler will use to resolve the types and assign the correct integer or floating point version of the Forth operation in the final output; these internal operations all have an underscore suffix. The first few equations are thus similar to those near (3.1).

$$P_E(e \frown \texttt{"+"} \frown t) = P_E(e) \frown P_T(t) \frown \texttt{"~~+\_"}$$

$$P_E(t) = P_T(t)$$

$$P_E(e \frown \texttt{"-"} \frown t) = P_E(e) \frown P_T(t) \frown \texttt{"~~-\_"}$$

$$P_T(t \frown \texttt{"*"} \frown f) = P_T(t) \frown P_F(f) \frown \texttt{"~~*\_"}$$

$$P_T(t \frown \texttt{"/"} \frown f) = P_T(t) \frown P_F(f) \frown \texttt{"~~/\_"}$$

$$P_T(f) = P_F(f)$$

$$P_F(\texttt{"("} \frown e \frown \texttt{")"}) = P_E(e)$$

Below this, terminals begin to be output in the intermediate code, however these need to be strings rather than literals; spaces are added where required to ensure the final output is consistently space-separated. In the equations below, the symbol $\boxed{\texttt{\textbackslash"}}$ denotes a literal embedded quote, and ϕ is a function identifier (not a numerical variable). We also have the additional parameters

n Any integer number string.

i Any integer identifier (variable) string — note, no longer a general identifier

l Any list of arguments.

r Any floating point number string.

r' Any Forth equivalent[1] to r.

x Any real identifier string (floating point variable).

The identifiers are stored in symbol tables which specify some predefined families of identifier strings; this obviates any immediate need for typed declarations, however this is an area which will be examined at a later stage. Thus identifiers beginning with the letters `i-n`, in either case, are integer while all others are treated as floating point.

We then introduce the additional parsing functions

P_{F0} Returns the intermediate code for an unsigned factor.

P_L Returns the intermediate code for an argument list.

[1] We distinguish these as they may use different symbols for unary minus in exponent notation.

The remaining conversion equations for the intermediate code can now be defined (note L is shorthand for l, e; and l could be a single e).

$$
\begin{aligned}
&P_F(\texttt{"}\sim\texttt{"} \frown f) = P_F(f) \frown \texttt{" NEGATE\_"} \\
&P_F(f) = P_{F0}(f) \qquad\qquad (f \text{ can be } n, i, r, x, \text{ or } \phi) \\
&P_{F0}(\phi(L)) = P_L(L) \frown P_{F0}(\phi) \qquad (P_{F0} \text{ would simply output the string } \phi) \\
&P_{F0}(n) = \backslash\texttt{"} \frown n \frown \backslash\texttt{"} \frown \texttt{" int"} \\
&P_{F0}(i) = \backslash\texttt{"} \frown i \frown \backslash\texttt{"} \frown \texttt{" int"} \\
&P_{F0}(r) = \backslash\texttt{"} \frown r' \frown \backslash\texttt{"} \frown \texttt{" float"} \\
&P_{F0}(x) = \backslash\texttt{"} \frown x \frown \backslash\texttt{"} \frown \texttt{" float"} \\
&P_L(l \frown \texttt{","} \frown e) = P_L(l) \frown P_E(e) \\
&P_L(e) = P_E(e)
\end{aligned}
$$

The final four P_{FO} equations show the type information being included in the output string; for example, the tokens “36” or “3.142” become the strings (with embedded quotes around the numbers)

“ " 36" int”
“ " 3.142" float”

This is because we are still in the intermediate code, which will be converted to input for Forth by the second-pass operations — those with the appended underscore — e.g. +_ , *_ . We can now examine these in the context of the second pass of the compiler.

4.2 Second Pass

For the simple example infix input 10.5+5$*$2.5, the first pass will have produced the intermediate output string with embedded quotes

" 10.5" float " 5" int " 2.5" float *_ +_ (4.1)

This is then tokenised and interpreted by the second-pass compiler. At this level, `int` and `float` are interpreted as integer constants, so when the operation *_ is reached, the stack has three string/integer pairs on it. The code for *_ is shown in figure 4.

It takes four arguments consisting of two string/integer pairs, the integers containing type information. The four possible combinations of `int` and `float` are checked (the second case is applicable here). The numerical/ variable values are output as strings, followed by the appropriate ordinary arithmetical operators, here either * or `F*` (the floating point version). In addition, since `F*` requires that both its arguments be floating point, the conversion operator[2] `S>F` is inserted in the output string after any integer values.

[2] Single-precision integer to Floating point.

```
: *_  ( az1 type1 az2 type2 -- az3 type3)
 (: VALUE e1  VALUE type1  VALUE e2  VALUE type2 :)
  CASE
   type1 int =  type2 int =  AND  ?OF
    e1 e2 ^ " *" ^ int
   ENDOF
   type1 int =  type2 float =  AND  ?OF
    e1 " S>F" ^ e2 ^  " F*" ^  float
   ENDOF
   type1 float =  type2 int =  AND  ?OF
    e1  e2 " S>F" ^ ^  " F*" ^  float
   ENDOF
   type1 float =  type2 float =  AND  ?OF
    e1  e2  ^  " F*" ^  float
   ENDOF
   ( otherwise ) " Type error" AZ-ABORT
  ENDCASE
 2LEAVE ;
```

Figure 4: RVM-Forth code for *_ .

Finally the constant corresponding to the value is output, either `int` or `float`. Thus the output is a string and an integer, suitable for input to another operation of this kind.

After *_ has interpreted and dealt with the relevant parts of string (4.1), we come to +_ , which finds the following four arguments on the stack:

```
"10.5" float "5 S>F 2.5 F*" float
```

the last two being the output from *_ . Note these strings are actual strings.

The code for +_ is very similar to the above, and the final output will be

```
"10.5 5 S>F 2.5 F* F+" float
```

The final type indicator is dropped from this (by the second-pass compiler when the end of the expression is reached), and the remaining string can now be compiled by Forth.

For an example with identifiers (subject to the conventions mentioned in 4.1, page 8) we use

$$(i + 7) * (j + 1.5)$$

The first pass produces the *string* (recall these quotes are embedded)

```
" i" int " 7" int +_ " j" int " 1.5" float +_ *_
```

After both +_ have been processed, the stack will contain the following:

```
"i 7 +" int "j S>F 1.5 F+" float
```

Finally *_ will produce this

```
"i 7 + S>F j S>F 1.5 F+ F*"
```

having dropped the `float` type indicator.

The second pass doesn't have such a neatly defined set of equations to encapsulate it, as the arguments are no longer a single string, but four arguments (stack parameters): string, integer, string, integer. Also, `NEGATE_` and `F->F_`, dealing with negating variables and numbers, and type-checking the arguments for function applications[3], have different signatures again. They also have more conditions to cover with, for instance, four combinations of `int` and `float`. The specifying equations for +_ would be:

$$+\_(n_1, \texttt{int}, n_2, \texttt{int}) = n_1 \frown n_2 \frown \texttt{" +"}$$

$$+\_(n, \texttt{int}, x, \texttt{float}) = n \frown \texttt{" S>F"} \frown x \frown \texttt{" F+"}$$

$$+\_(x, \texttt{float}, n, \texttt{int}) = \texttt{" S>F"} \frown x \frown n \frown \texttt{" F+"}$$

$$+\_(x, \texttt{float}, x, \texttt{float}) = \texttt{" S>F"} \frown x \frown \texttt{" S>F"} \frown x \frown \texttt{" F+"}$$

4.3 Floating Point State Machine

For the general numeric literals subsystem — which handles floating point and exponent form literals — we adopt a state machine. A feature of the exponent form for infix expressions is that unary minus in an *exponent* must be written as a tilde rather than a normal minus (this does not apply to any other unary minus). Thus expressions such as

`-3.467e~6` or `-2.5e~10`

are admissible. This is to simplify the otherwise unwieldy process of identifying unary minuses for floating-point exponents and replacing them internally with tildes; the outputs will have normal minus signs for the usual Forth input. The machine itself has seven states; N_2, N_4 and N_7 are terminal states, and ϵ denotes an empty string.

[3] Neither are described in detail here.

$N ::= N_1 \mid DN_2 \mid .N_3$

$N_1 ::= DN_2 \mid .N_3$

$N_2 ::= \epsilon \mid DN_2 \mid .N_4 \mid eN_5$

$N_3 ::= DN_4$

$N_4 ::= \epsilon \mid DN_4 \mid eN_5$

$N_5 ::= N_6 \mid DN_7$

$N_6 ::= DN_7$

$N_7 ::= \epsilon \mid DN_7$

5 Conclusions and Further Work

The idea for this course stemmed from a desire to teach Language Systems at a more pure conceptual level, so that the concepts would be less obscured by their almost immediate abandoning for a more complex bottom-up approach, as was previously done.

We therefore used a top-down right-to-left method to parse expressions down to factor level, using recursive descent techniques to generate postfix versions readable by the Forth system — later via intermediate code to hold type information. The course thus began with deceptively simple concepts and could build naturally to a reasonably capable two-pass expression compiler. So the students learn about grammars and programming simple compilers in Forth.

We could go in a number of different directions with this, perhaps expanding in other ways than the floating point facility; for instance, one idea is to adapt this approach to make use of RVM Forth's native support for sets (using the C package contributed by Frank Zeyda [5]). This could be used to develop, for the course, an application to accept sets and basic set operations and generators from a suitably defined grammar, and convert them to valid Forth. Even eschewing floating-point support here, we still have the issue of types, as sets can contain strings and integers, and also pairs (maplets, using the ASCII notation |->) and nested sets. Therefore recursion would again be required to tease out the type information at each level and generate the appropriate tags for the first-pass output.

For illustration, a brief example of a simple set enumeration with string-integer pairs and the output which would be generated:

```
{"Dave" |-> 3291, "Li" |-> 3419} becomes
STRING INT PROD { " Dave" 3291 |-> , " Li" 3419 |-> , }
```

The grammar would, among other things, have to ensure that the maplet operator retained left associativity, and had the correct type signature. Some operators have right associativity, for instance domain restriction, and the bidirectional functionality will be needed to parse these. We will give consideration to a generalised and consistent system for type declarations in these language grammars, which would work with the set types of Forth and also with numeric types; also a closer analysis of the time complexity over a greater variety of grammars.

References

[1] A. V. Aho and J. D. et al Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2nd edition, 2006.

[2] R. C. Backhouse. *Syntax of Programming Languages: Theory and Practice*. Prentice-Hall, 1979.

[3] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM, Volume 13, No 2*, 1970.

[4] W. J. Stoddart and A. R. Lynas. A Reversible Computing Approach to Forth Floating Point. In M. A. Ertl and P. Knaggs, editors, *23rd EuroForth Conference Proceedings*, October 2007. On-line proceedings.

[5] W. J. Stoddart and F. Zeyda. Implementing Sets for Reversible Computation. In M. A. Ertl, editor, *18th EuroForth Conference Proceedings*, 2007. On-line proceedings.

The RVM-Forth software and manual can be downloaded from

```
http://www.scm.tees.ac.uk/formalmethods/download/rvm0_1.zip
http://www.scm.tees.ac.uk/formalmethods/download/rvmman.pdf
```

Java Framework for Static Analysis of Forth Programs

Jaanus Pöial

The Estonian Information Technology College
e-mail: `jaanus.poial@itcollege.ee`

Abstract. In [2] author introduces theoretical background for static analysis of Forth programs (definitions, basic operations, typing rules, etc.). This paper is direct continuation of the topic and describes implementation of basic blocks for writing software tools to support type checking of Forth programs. On small examples we try to explain problems and possible solutions. Author hopes that these ideas help to develop some useful tools. Prototype is written in Java that is quite universal and widespread object oriented platform for software development.

Keywords: *Type Systems, Forth, Program Analysis*

1 Introduction

In [1] we first defined formal stack effects of Forth words. This definition and theory of stack effect calculus have been developed for a long time and in [2] we introduced several operations on effects to perform static type analysis of Forth texts. Basic program constructs covered so far are sequence, choice and iteration. For each of these we have corresponding operation in our calculus.

Let us start with a few informal examples. Data item on Forth stack generally does not have any run-time type but the programmer usually has some static type information in mind when composing a program. This information may be more or less exact, e.g.
`a-addr < c-addr < addr < u < x` (`x` is the least exact)
and for example word `@` is specified as `(a-addr -- x)`.

At the same time many operations manipulate stack (e.g. `SWAP, DUP, ROT,` etc.) without changing types of data items. To cover this aspect we introduced position indices to type symbols:
word `SWAP` has effect `(x[2] x[1] -- x[1] x[2])` rather than `(x x -- x x)`.
Program `DUP @` has stack effects `(x[1] -- x[1] x[1])(a-addr -- x)` and should evaluate into `(a-addr[1] -- a-addr[1] x)` rather than `(x -- x x)`.

Sequences can be longer than two words, e.g. `SWAP DUP @` gives:
`(x[2] x[1] -- x[1] x[2])(x[1] -- x[1] x[1])(a-addr -- x)`.
When two type symbols with locally defined indices must match in the process of evaluating a sequence they produce a new type symbol that has minimal (most

* Supported by Estonian Science Foundation grant no. 6713

exact) type and new "fresh" index. This new symbol replaces both matching symbols in the sequence:
(x[2] x[1] -- x[1] **x[2]**)(**x[1]** -- x[1] x[1])(a-addr -- x)
(x[3] x[1] -- x[1] x[3])(x[3] -- x[3] **x[3]**)(**a-addr** -- x)
(a-addr[4] x[1] -- x[1] a-addr[4])(a-addr[4] -- a-addr[4] a-addr[4])
(a-addr[4] -- x)
Let us rename (to delete unused) indices where possible:
(a-addr[2] x[1] -- x[1] a-addr[2])(a-addr[2] -- a-addr[2] a-addr[2])
(a-addr[2] -- x)
and the final evaluation result for sequence SWAP DUP @ is
(a-addr[2] x[1] -- x[1] a-addr[2] x).

On this small example we see that evaluation has to preserve information both on types and positions of data items. When type symbols do not match we have to produce some useful error information. This is one of the reasons to have evaluation of sequence as a basic block in our framework rather than composition of two effects.

Choice between two branches of a program in our framework forces these branches to have "the same" effect. Operation *glb* (greatest lower bound) of two effects tries to match all corresponding symbols and replace these with new most exact "fresh" symbols (like for composition above).

Program IF ! ELSE C! THEN has two branches and we calculate
glb((x a-addr --),(char c-addr --)*)* as (char a-addr --).

Program IF OVER ELSE @ DP @ THEN produces
glb((x[2] x[1] -- x[2] x[1] x[2]),(a-addr -- x x)*)* that evaluates into
(x[2] a-addr[1] -- x[2] a-addr[1] x[2]).

From these examples we conclude that *glb* calculates longest type lists with most exact types.

Iteration in this framework forces the loop body not to change the stack state (loop body has "idempotent" effect: e = ee).
Effects with equal type lists on both sides are idempotents: (*list* -- *list*).
To calculate the effect of a loop we find the most precise idempotent by matching left and right sides of the effect that describes the loop body (it is possible only if both sides have the same length).

Program BEGIN @ AGAIN iterates the word @ endlessly. The loop body has effect (a-addr -- x) and loop (as a whole) has effect
(a-addr[1] -- a-addr[1]).
More complicated loop BEGIN SWAP OVER WHILE NOT REPEAT falls into two pieces:
(x[2] x[1] -- x[1] x[2])(x[2] x[1] -- x[2] x[1] x[2])(flag --) and
(x -- x).
First loop body has effect (x[2] flag[1] -- flag[1] x[2]) and the loop has effect (flag[1] flag[1] -- flag[1] flag[1]). Composed by the second loop effect (x -- x) we still have (flag[1] flag[1] -- flag[1] flag[1]) but we also know that NOT operates on flag[1].

If the loop body has effect *e* we can calculate loop effect as *glb(e, ee)*.

These examples are not complex enough to cover real programs but hopefully give some ideas how to evaluate sequences, choices and iterations.

2 Java framework

Package `evaluator` consists of several classes and probably will grow depending on tools we intend to develop. Let us summarize the basic blocks in this framework.

Class `TypeSymbol` defines symbolic type of a stack item (like `a-addr, flag, char, ...`) together with position index (used by stack manipulation words like `SWAP, OVER, ROT, ...`). Type names must be known by current typesystem. Usually there are more names than actual types (synonyms are allowed for convenience). Position indices are integers (when ”fresh” symbol is created during the match operation this index increases, index 0 is used if position is not important).

Class `TypeSystem` is used to define and query subtyping relations between types. Type name is used as a key to access matrix of relations. Relations are `"incompatible", "subtype", "supertype", "synonym"`. Typesystem is static, once created it does not change much during evaluation process. But we keep possibility to use different typesystems to analyze the same program open (e.g. to see more or less details).

Class `Tvector` describes the stack state (top right) and each vector consists of typesymbols. Substitution of one symbol by another is defined in this class.

Class `Spec` describes the stack effect (specification). It consists of two vectors (left side - stack state before execution, right side - stack state after execution) and additional workfields (e.g. string read by scanner words like `."` or `(`). Major operations of the framework (like greatest lower bound or finding idempotent for the loop body) are defined in this class:
`spec1.`**glb**`(spec2, typesystem, specset) returns spec`
`bodyspec.`**idemp**`(typesystem, specset) returns loopspec`

Class `SpecSet` describes a mapping from Forth words to stack effects. This mapping is dynamic - all new words defined in the program must be added. Once again, we may use different specsets for the same program text to analyze different aspects (e.g. run-time stack vs. compile-time stack).

Class `SpecList` describes a linear sequence of stack effects and implements evaluation of this list against given typesystem and given specset:
`speclist.`**evaluate**`(typesystem, specset) returns spec`
Composition of stack effects is a particular case of evaluation.

Class `ProgText` is inner representation of the Forth program we want to analyze. Currently only linear sequence of words is implemented.

Class `Evaluator` contains the main method and a small demo that adds annotations (comments about stack state) to the linear program text.

3 Further work

All classes described above are prototypes and need to be implemented fully to develop any useful tools. Extensible nature of Forth demands that our framework is also extensible (for example, we cannot just fix control structures or data definition words). It seems to be easier to re-write this package in Forth to achieve full extensibility.

Stack effect calculus might help programmers when integrated into Forth IDE (e.g. editor that shows current stack state symbolically, evaluates selected text, etc.). To create such an editor we need these (or similar) basic blocks plus a lot of other components. Maybe it is reasonable to use some existing IDE platform (like Eclipse - `www.eclipse.org`) and develop a Forth plugin for Eclipse. Then probably we lose in extensibility but we can still work with some subset of Forth. The following is a very small but useful subset that we would like to cover next:

```
PROG = ELEM / PROG ELEM .
ELEM = SIMPLE / DEFINITION .
SIMPLE = WORD / PARSER / CONSTANT .
WORD = <word> .
PARSER = PARSER<delim> / COMMENT .
PARSER<delim> = WORD <string><delim> .
COMMENT = <comment> .
CONSTANT = <constant> .
DEFINITION = VARDEF / CONSTDEF / COLONDEF .
VARDEF = 'VARIABLE' NAME / 'CREATE' NAME .
CONSTDEF = SIMPLIST 'CONSTANT' NAME .
COLONDEF = ':' NAME CONTENT ';' /
':' NAME CONTENT 'CREATE' CONTENT 'DOES>' CONTENT ';' .
NAME = <word> .
CONTENT = CELEM / CONTENT CELEM / .
CELEM = SIMPLE / STRUCTURE .
STRUCTURE = 'IF' CONTENT 'THEN' /
'IF' CONTENT 'ELSE' CONTENT 'THEN' /
'BEGIN' CONTENT 'WHILE' CONTENT 'REPEAT' /
'[' SIMPLIST ']' .
SIMPLIST = SIMPLE / SIMPLIST SIMPLE .
```

References

1. Pöial J., "Algebraic Specifications of Stack Effects for Forth Programs," *1990 FORML Conference Proceedings, EuroFORML'90 Conference, Oct 12 – 14, 1990, Ampfield, Nr Romsey, Hampshire, UK,* Forth Interest Group, Inc., San Jose, USA, p. 282 – 290, 1991.
2. Pöial J. "Typing Tools for Typeless Stack Languages," *Proc. 22-th EuroForth Conference, September 15 – 17, 2006, Cambridge* p. 40 – 46, 2006.

Updating the Forth Virtual Machine

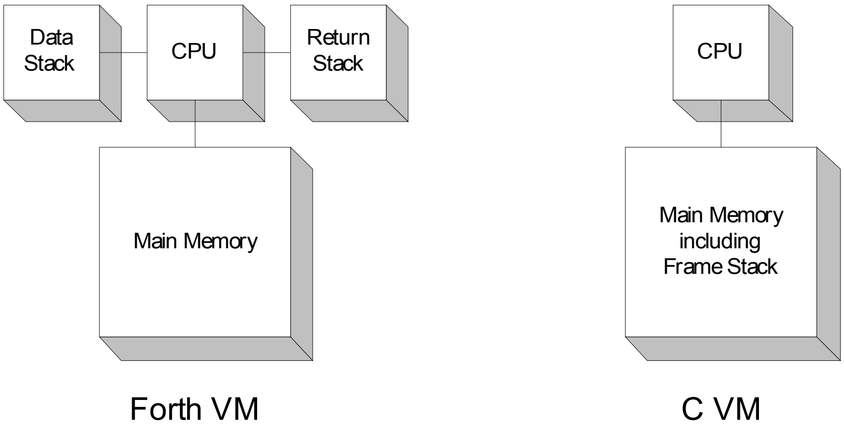
Stephen Pelc
MicroProcessor Engineering
133 Hill Lane
Southampton SO15 5AF
England
tel: +44 (0)23 8631 441
fax: +44 (0)23 8033 9691
net: sfp@mpeforth.com
web: www.mpeforth.com

Abstract

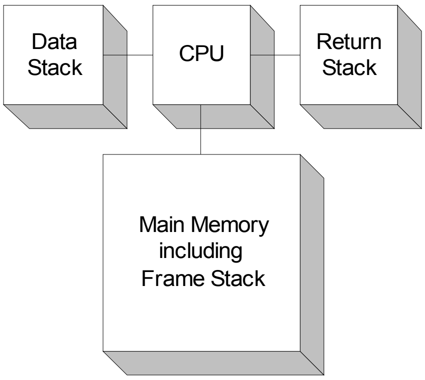
The canonical Forth Virtual machine has remained essentially the same since its inception. Modern silicon implementations and compiler techniques indicate that the VM as used in practice differs from this model. It is time to consider overhauling the canonical Forth Virtual Machine. In particular, the addition of address registers is considered.

Introduction

Classical or canonical Forth views the world as a CPU connected to main memory and two stacks which are not addressable, and are quite separate from main memory. C views the world as a CPU connected to memory, which includes a list of frames (usually a stack of frames) which **must** be in addressable memory.



By adding the necessary registers for the frame stack to the canonical Forth machine, we arrive at the basic design of the SENDIT VM, which was discussed in various papers in the late 1990s. SENDIT (EP9152) was a project carried out under the European Union's ESPRIT research and development programme. SENDIT was based upon the results of a preceding project, PROCIC EP5497, and produced tools for the development of heterogeneous networks for use in embedded and real time applications.



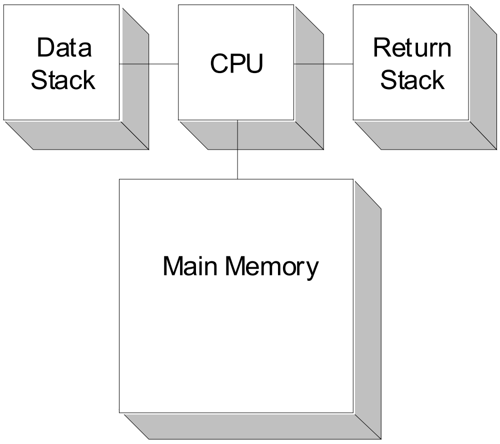
PC = program counter
PSP = parameter stack pointer
RSP = return stack pointer

FP = frame pointer
FEP = frame end pointer

SENDIT VM and registers

The SENDIT VM looks remarkably similar to other stack machine CPUs derived from a Forth architecture and designed to execute C efficiently.

Another branch of the Forth virtual machine has been called machineForth, and appears in software implementations such as ColorForth and various CPUs from iTV, Ultratechnology and IntellaSys, most lately in the SEAForth S24 multicore chips.



PC = program counter
PSP = parameter stack pointer
RSP = return stack pointer

A = A index and scratch
B = B index and scratch

machineForth VM and registers

Other CPU core designs include MicroCore and designs from Bernd Paysan, Brad Eckert and Chris Bailey.

What distinguishes these cores is that they introduce data cells, registers and operations that are unsupported by the canonical Forth machine. In the description I have chosen not to include the TOS, NOS and TOR virtual registers. TOS and NOS are common across virtually all implementations as ALU inputs and outputs. TOR has wide variation in implementation for anything other than to hold a return address.

This paper explores the impact of these designs on how the Forth programming language could be changed.

Why update the Forth Virtual Machine?

The canonical Forth virtual machine is weak in several areas.

1) It does not execute C well, which is important for commercial exploitation of silicon stack machines.
2) It is weak for DSP operations, which restricts performance in embedded applications without changes to the VM or much increased compiler complexity,
3) Without index operations, it is cumbersome to deal with complex data structures whose base address is passed as an argument to a word.

Execution of C requires a frame pointer for access to local variables and buffers.

DSP operations often require three or four parameters to be manipulated regularly, e.g.

1) source address, destination address and length,
2) first source address, second source address, destination address and length.

Canonical Forth requires ugly source code to deal with these situations. Silicon implementations such as C18, FR32 and the Teesside University machines have provided index and scratch registers, whereas others have provide more access to the top of the return stack. Using the top of the return stack as a loop counter has been common for some time, e.g. the **`FOR ... NEXT`** loop structure.

The Forth community has long talked about TOS (top of data stack), NOS (next/second on data stack) and TOS (top or return stack). These are not quite enough for DSP operations an Chuck Moore's current silicon includes A and B registers which are used both as index registers and for scratch storage.

A new Forth Virtual Machine

I claim no particular novelty in this machine. It is a synthesis of practice that has been observed in several software and silicon machines over the years. What triggered this paper was seeing that Forth various compilers, e.g. Gary Bergstom's AFT (Another Forth Translator) have either implemented additional registers and facilities in their Forth VMs, or are seriously considering doing so.

If we look at what is common between these designs we find the following that can be treated as registers rather than just as ALU connections.

A	Register used as a scratch or index register, often with auto-increment and/or auto-decrement addressing.
B	Register used as a scratch or index register, often with auto-increment and/or auto-decrement addressing.
LP	Local frame pointer with base+literal indexed addressing.
UP	User area pointer with base+literal indexed addressing for thread-local storage.

Table 1: Additional Forth VM registers

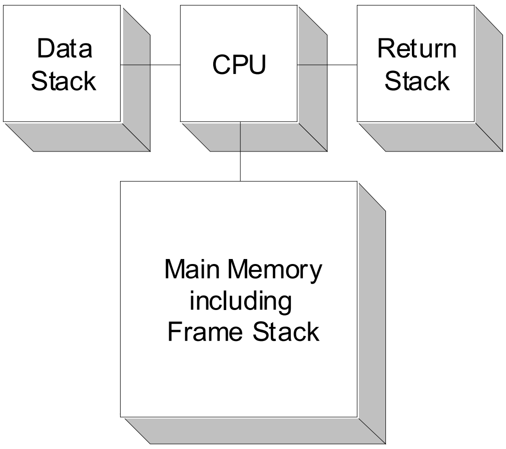
Inspecting various Forth implementations and source code, we can make various observations:

1) Use of the A and B registers considerably reduces the need for local variables.
2) Use of the A and B registers can considerably reduce stack manipulation in both source and compiled code.
3) Although UP can be implemented as a variable, most Forth systems, especially embedded systems, implement it using a CPU register.
4) What distinguishes the A/B pair and the LP/UP pair is that A/B implement auto-increment addressing, and occasionally auto-decrement addressing. The LP/UP pair implement base + offset addressing.
5) The use of the scratch registers improves source code density (level of abstraction) and reduces stack shuffling at basic block boundaries and avoids complexity in code generators.

In order to avoid mandating use of these registers, we can simply rename them in terms of how they are used:

A	Register used as a scratch or index register, often with auto-increment and/or auto-decrement addressing.
B	Register used as a scratch or index register, often with auto-increment and/or auto-decrement addressing.
X	Memory pointer with base+literal indexed addressing.
Y	Memory pointer with base+literal indexed addressing.

Table 2: Additional registers in the new VM



PC = program counter
PSP = parameter stack pointer
RSP = return stack pointer

A = A index and scratch
B = B index and scratch
X = index
Y =index

A possible new Forth VM and registers

Wordsets

A and B registers

This a fully featured wordset. Some systems only provide auto-increment/decrement on the A register. On some systems, the B register cannot be read. The A and B registers provide the source and destination address pointers used for block, string and DSP operations as well as providing scratch storage.

`>A` `\ x --`
Writes to the A register.

`>B` `\ x --`
Writes to the B register.

`A>` `\ -- x`
Reads the A register.

`B>` `\ -- x`
Reads the B register.

`A@` `\ -- x`
Read the memory pointed to by the A register.

`A!` `\ x --`
Write the memory pointed to by the A register

`B@` `\ -- x`
Read the memory pointed to by the B register.

`B!` `\ x --`
Write the memory pointed to by the B register

`A@+` `\ -- x`
Read memory pointed to by A, increment A by one cell. A post-incremented read.

`B@+` `\ -- x`
Read memory pointed to by B, increment B by one cell. A post-incremented read.

`A@-` `\ -- x`
Read memory pointed to by A, decrement A by one cell. A post-decremented read.

`B@-` `\ -- x`
Read memory pointed to by B, decrement B by one cell. A post-decremented read.

`A!+` `\ x --`
Write to the memory pointed to by A, and update A.

`B!+` `\ x --`
Write to the memory pointed to by B, and update B.

X and Y registers.

The X and Y registers provide indexed addressing. In Forth they can be used to implement the **`USER`** area and local frame pointers.

`>X` `\ x --`
Writes to the X register.

`>Y` `\ x --`
Writes to the Y register.

`X>` `\ -- x`
Reads the X register.

`Y>` `\ -- x`
Reads the Y register.

`nX@` `\ n -- x`
Read the memory pointed to by the X register plus n (literal) address units.

nX! \ x --
Write the memory pointed to by the X register plus n (literal) address units.
nY@ \ -- x
Read the memory pointed to by the Y register plus n (literal) address units.
nY! \ x --
Write the memory pointed to by the Y register plus n (literal) address units.

Biquad filter example

My thanks go to Gary Bergstrom for permission to publish this code.

```
: *.       \ fr1 fr2 -- fr3
\ Fractional multiply.
  +1. */  ;
: 1STEP+   \ sum -- sum'
\ Perform a multply/accumulate step, incrementing both
\ pointers.
  B@+ A@+ *. +  ;
: 1STEP-   \ sum -- sum'
\ Perform a multply/accumulate step, incrementing the
\ coefficient pointer and decrementing the data pointer.
  B@+ A@- *. +  ;
: SHIFT2   \ fr --
\ The last step of the filter. The current data item
\ is shifted into the next data slot and replaced by fr.
  A@ SWAP A!+ A!+ ;
: (BIQUAD) \ frx -- fry
\ The core of the biquad filter operation.
  DUP >R
  B@+ *.              ( initial sum = B0*input )
  1STEP+ 1STEP-  R> SHIFT2
  1STEP+ 1STEP-  ;
: BIQUAD   \ fx addr-filt addr-coef -- fry
\ A single order biquad filter.
  >B >A  (BIQUAD) DUP SHIFT2  ;
: 2xBIQUAD \ fx addr-filt addr-coef -- fry
\ A second order biquad filter.
  >B >A (BIQUAD) (BIQUAD) DUP SHIFT2  ;
```

References and further reading

[1] SENDIT token specification, ISBN 0-9525310-1-1, MicroProcessor Engineering, 133 Hill Lane, Southampton, England

[2] Europay Open Terminal Architecture
Volume 1 - Token Specification
Volume 2 - Forth language binding
Volume 3 - C language binding
MasterCard International, 198A Chaussée de Tervuren, 1410 Waterloo, Belgium

[3] The evolution of SENDIT into EPIC, Stephen Pelc, Rochester Forth Conference 1996

[4] The SENDIT project: a Forth in sheep’s clothing, Jon Lee, Rochester Forth Conference 1996

[5] SENDIT tool architecture, ISBN 0-9525310-2-X, MicroProcessor Engineering, 133 Hill Lane, Southampton, England. Out of print.

[6] MicroCore:
http://www.microcore.org

[7] Stack Computers: the new wave, Philip J. Koopman, Jr.,
http://www.ece.cmu.edu/~koopman/stack_computers/index.html

[8] IntellaSys SEAforth-24,
http://www.intellasys.net

[9] C18, Chuck Moore,
http://www.complang.tuwien.ac.at/anton/euroforth/ef01/

[10] B16, Bernd Paysan,
http://www.jwdt.com/~paysan/b16.html

[11] AFT, Gary Bergstrom
private communication

A Debugger for the b16 CPU

Bernd Paysan

EuroForth 2008, Vienna

Abstract

A debugging interface for the b16 CPU is shown. A few lines of Verilog and some small Forth programs are sufficient to add a classical debugging interface to this small CPU. Integration of other controls (like test equipment) is very easy to do.

1 Motivation

For the current project with the b16 core [1] inside, a few things are “unusual”:

- Firmware programmer isn’t a Forth expert (i.e. not me)
- Program in writable memory (first test chip: RAM, final chip: Flash or OTP)

Under these circumstances, it makes some sense to debug the firmware using a “classical” in–circuit–debugger. It will turn out that adding such a debugger to the hardware is a fairly trivial exercise, leaving writing the software as “main” challenge.

The features such a debugger should have are quite common:

- Interface the chip with a PC, so that the PC can control memory content (and memory mapped IO registers)
- The debugging window should show the source code, and jump with the cursor to the currently executed location (if the CPU is halted)
- Typical commands: Single step, multiple steps, run/stop, set/clear breakpoint
- Direct access to a memory location, dump of a consecutive memory block
- Optional: Forth console to mix debugging commands with other instructions (e.g. measurement and stimuli equipment driven by serial lines)

What’s missing

- Classical command line for the embedded CPU

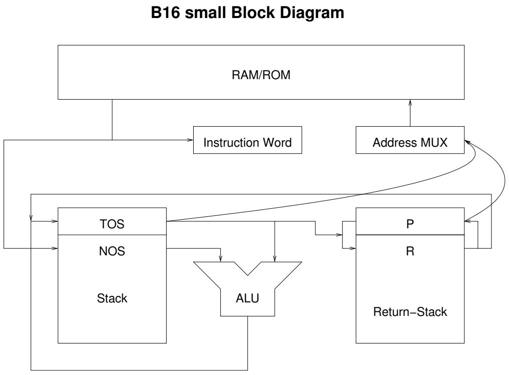


Figure 1: Block Diagram

1.1 Architectural Overview

Just to recap: The core components of the b16 are

- An ALU
- A data stack with top and next of stack (T and N) as inputs for the ALU
- A return stack with top R
- An instruction pointer P
- An instruction latch I

Figure 1 shows a block diagram.

2 Adding In–Circuit Debugging

From a previous project, we already had two important parts: The CPU in a shape that’s useful for the project (multiplication and division, dropped back then, were added again), and a SPI–derived interface to directly access memory from outside. The interface uses only two pins, by sharing DI/DO, and interpreting activity on the clock line as chip select (with timeout). The device is not pad limited, but the package gets cheaper with less pins; a standard SPI interface that allows tristating DO can talk to this chip without problems. To interface with the PC, an FTDI module is used (bit–banging mode of serial port interface).

So the missing link was the actual debugger.

The registers were implemented in the order described here. This turned out as a not quite clever idea, but it was possible to work around the problem. The SPI interface can read multiple words in one go, by incrementing the address latch after each read access. This has the side effect that each read sequence ends with a read to the next memory location, even if this data is never used (it just has to be available on the next rising clock edge).

2.1 Implementation

For debugging purposes, all registers are memory read–writable. This requires an external bus master attached to the debugging interface. It's only active when the processor is stopped, so the processor itself can't access its own registers.

The debugging module offers the following registers as address space:

Address	*read*	*write*
$FFE0	P	P
$FFE2	T	T
$FFE4	R	R
$FFE6	I	I
$FFE8	state	state
$FFEA	stack[sp]	push+T
$FFEC	rstack[rp]	pushr+R
$FFEE	stop	start/step

The address $FFEE is special, since a read access to it stops the CPU. By writing to $FFEE, the debugger can either continue the program (write 1 there), or cause it to single step (write 0 there).

⟨*debugging read* 2a⟩≡

```
reg ‘L dout;

always @(daddr or dr or run or
         P or T or R or I or
         state or sp or rp or c)
if(!dr || run) dout <= 'hz;
else casez(daddr)
   3'h0: dout <= P;
   3'h1: dout <= T;
   3'h2: dout <= R;
   3'h3: dout <= I;
   3'h4: dout <= { run, 4'h0, c, state,
                   {4-sdep{1'b0}}, sp,
                   {4-rdep{1'b0}}, rp };
   3'h5: dout <= N;
   3'h6: dout <= toR;
   3'h?: dout <= 0;
endcase
```

⟨*debugging-ports* 2b⟩≡

```
   input [2:0] daddr;
   input dr, dw;
   input ‘L din;
   output ‘L dout;
```

⟨*debugging* 2c⟩≡

```
if(dw) casez(daddr)
   3'h0: P <= din;
   3'h1: T <= din;
   3'h2: R <= din;
   3'h3: I <= din;
   3'h4: { c, state, sp, rp } <=
           { din[10:8],
             din[sdep+3:4], din[rdep-1:0] };
   3'h5: { sp, T } <= { spdec, din };
   3'h6: { rp, R } <= { rpdec, din };
endcase
if(dr) casez(daddr)
   3'h5: sp <= spinc;
   3'h6: rp <= rpinc;
endcase
```

⟨*debugger* 2d⟩≡

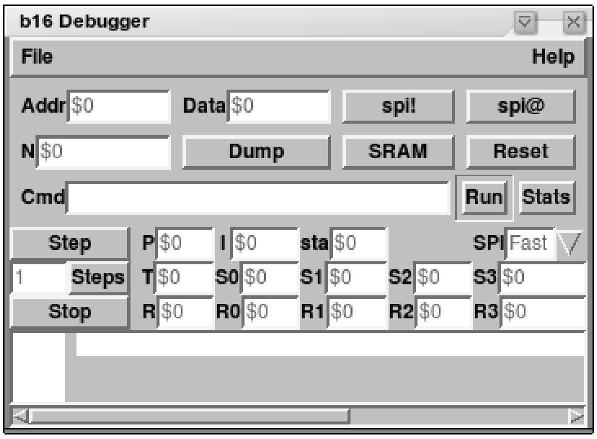
```
module debugger(clk, nreset,
                addr, data, r, w,
                drun, dr, dw);
parameter l=16, dbgaddr = 12'hFFE;
input clk, nreset, r;
input [1:0] w;
input ‘L addr, data;
output drun, dr, dw;

reg drun, drun1;
wire dsel = (addr[l-1:4] == dbgaddr);
assign dr = dsel & r;
assign dw = dsel & |w;

always @(posedge clk or negedge nreset)
if(!nreset) begin
   drun <= 1;
   drun1 <= 1;
end else begin
   drun <= drun1;
   if((dr | dw) && (addr[3:1] == 3'h7)) begin
      drun <= !dr & dw;
      drun1 <= !dr & dw & data[0];
   end
end

endmodule
```

⟨*dbg senselist* 2e⟩≡

```
or run or dw or daddr
```

⟨*stack debugging* 2f⟩≡

```
if(!run && dw) casez(daddr)
   3'h5: dpush <= 1;
   3'h6: rpush <= 1;
endcase
```



Addr $0 Data $0
N $0
P $0 I $0 sta $0 SPI Fast
1 Steps T $0 S0 $0 S1 $0 S2 $0 S3 $0
R $0 R0 $0 R1 $0 R2 $0 R3 $0

Figure 2: Debugging GUI

3 Debugging Software

The debugging GUI is just a MINOS window which shows the main states and opens a window to the source code (see figure 2).

The SPI interface code was already available from the last project [2] — it needed slight changes, though. First of all, there were two different bit orders of SPI interfaces availbable, both with consistent test environments, and the coworker in charge picked the little endian one[1].The SPI post–access read had a bad effect on the CPU status register read: This would also read the data stack, and as side effect increment the stack pointer. No good idea, especially, since the CPU status register also tells you if the CPU is running or halted by the debugger. A reordering of these registers would be a good idea.

What's worse is that the instruction register content changes the side–effect: Only NOPs really increment the stack pointers, other instructions may interfere with the commands from the debugger. So the workaround found was to first read the four registers P, I, T, and R, and then write back 0 (all NOPs) into the I register. This then allows to read the status plus the two stacks, and then again the two stacks until a full wrap–around of the stack pointers is achieved. Finally, restore the original content of the instruction register.

⟨*read registers* 3⟩≡

```
: load-regs ( - )
  DBG_P regs 4 spiw@s
  0 DBG_I spiw!
  \ clear instruction register to read stacks
  DBG_STATE regs 8 + 3 spiw@s
  stack 16 + stack 4 + DO
      DBG_S[] I 2 spiw@s
  4 +LOOP
  regs 6 + w@ DBG_I spiw! ...
```

3.1 Breakpoints

The original idea how to implement breakpoints was to call the debugger status register. This plan was sabotaged by eliminating loops in the design, so the debugger status register is not accessible by the CPU itself. It won't halt the CPU then, as well. However, it turned out that this idea had been bad for another reason, as well: Calling the debugger status register wastes precious return stack space (20%!), and is not necessary at all. Instead, it's completely sufficient to replace the instruction where you want to break with an empty loop, and check the P register for the breakpoint addresses. The likelyhood that the CPU will be executing the current breakpoint is quite high under these circumstances (however, it's only 1/2, in the other case, the P register points to the next instruction).

If the debugger sees that a breakpoint has been reached, it will stop the CPU. It then has to single–step until the right state is reached (just before loading the instruction). For further execution in single–step mode, the original memory content is restored; only when the CPU goes to "run" mode, the breakpoints have to be restored (run from a breakpoint location then is done by single–stepping to the state where the instruction register has been loaded, then the effect of replacing that instruction by a "breakpoint" loop will not be recognized). Since empty loops are not possible at the last address of a 1k word block, the "workaround" is not fully functional. So far, the firmware is clearly below this 1k words total size limit, anyway.

3.2 Source Window

Looks fairly trivial: Just use the MINOS editor component, and load the source. Next to the editor component, there's a canvas, which can draw the addresses (obtained from the listing), and by clicking on an address, you set/clear a breakpoint. What's a bit less trivial was changing the assembler so that the listing contains meaningful information about the relation between cursor position and address+state of the CPU. So far, there are still a few bugs: `.org` statements don't write out the address into the listing stream (so that the start of the first instruction is not tagged), and the assembler doesn't expand tabs, while the editor window does. So code with tabs will not have the cursor at the right spot.

The source window currently is not an IDE window, i.e. changes won't result in anything. Adding this feature is possible, also adding the feature to automatically reload the source after it has been changed by another editor. However, with the typical Unix environment, people are happy to rerun assembler and restart the debugger after changes in

[1] Certainly, this sort of software gets written short before the chip arrives from the fab — during device debugging, the proof that this software could be written is sufficient.

the source code. Remember: The user isn't expecting this kind of magic anyway, so don't deliver.

4 Integrating other Testing Equipment

After successful deployment of the debugger, coordination between it and other test equipment has been needed quite soon. E.g. to characterize the ADC, you'll want to load a test program that loops ADC conversions and stores them into RAM, and force a certain voltage into the input pin; iterate over this process through the entire input voltage range. Now we are happy that our debugger is nothing but a simple Forth program, and all you need is to add a few other simple Forth words to drive HP instruments over RS232 (nowadays using some USB to serial converters, optimally not from FTDI, to avoid conflicts). Warning: Confusion may arise when you reboot the machine or replug the USB adapters, because the number scheme of USB serial ttys is first come, first serve type. Unfortunately, Intel also forgot to specify a unique per–device ID for this kind of device, so you can't use an alternative naming scheme.

5 Lessons Learned

After a few days work, this debugger was satisfying the "customer" (the coworker doing firmware development and me doing other device testing). It's a fairly trivial program, and the hardware behind is also fairly trivial; trivial enough that the gate count is insignificant. One wonders why in earlier days CPUs with in–circuit debuggers used to be quite expensive; also the cost for an debugger (hardware plus software plus IP) for traditional 8051 clones still is very high — even though that's a product that doesn't go into just one device.

This is a fairly simple approach; for the final device, the breakpoint mechanism e.g. won't work; at least when it's in OTP (and reflashing entire sections to just add a breakpoint is also no good idea). So in the final device, there will be a fairly limited set of breakpoint address registers and comparators.

- If time permits, diverging modules like the SPI should be merged and made configurable
- The register order should be changed so that the stack access doesn't require special care (stack access first)
    - Read with side effect is evil, anyway
- Integrating the assembler into the debugger should be fairly trivial, and thereby it creates an IDE with little effort
- Further magic could allow to seamlessly insert code with just a small stop and restart of the CPU
- Adding some (further) interactivity with the target CPU is also fairly trivial
- Hot–plugged devices must have a unique serial ID (this is a hint to Intel!!!)

References

[1] EuroForth 2004, *b16-small — Less is More*, Bernd Paysan

[2] EuroForth 2007, *Audio GUI: MINOS@work,* Bernd Paysan

Cleaning up after yourself

M. Anton Ertl[*]
TU Wien

Abstract

Performing cleanup actions such as restoring a variable or closing a file used to be impossible to guarantee in Forth before Forth-94 gave us `catch`. Even with `catch`, the cleanup code can be skipped due to user interrupts if you are unlucky. We introduce a construct that guarantees that the cleanup code is always completed. We also discuss a cheaper implementation approach for cleanup code than using a full exception frame.

1 Introduction

A frequent programming problem is to restore some state, free a resource, or perform some other cleanup reliably. Typical examples are:

- Restore `base` after a temporary change.
- Close a file.

In Forth-94 we can use `catch` to ensure that such cleanup actions happen under most (but not all) circumstances.

In this paper we explore ways to improve on this state of affairs in the following ways:

- Provide a more reliable mechanism that works even in the presence of asynchronous exceptions (e.g., user interrupts).
- Avoid the cost of a full-blown exception frame where possible.

2 Running Example

As a running example, we will use a word `hex.` that prints a number in hex base without changing `base`. And that word will be used in the following context:

```
: foo
  ... hex. ...
  ... .    ... ;

decimal foo
hex foo
```

Note that, in addition to printing a number in hex, `foo` also prints a number in the current base.

3 Standard Forth solutions

3.1 Thinking Forth approach

In the old days, Forth did not have `catch`, so one would write, e.g.:

```
: hex. ( u -- )
  base @ >r
  hex u.
  r> base ! ;
```

But even in the old days Forth had non-local exits via `abort`, and `quit`, as well as user interrupts. If any of these non-local exits from `u.` happened, `base` would not be restored.

So, in the old days, cleanup could not be performed reliably. So, various ways to work around this situation were developed and practised, as discussed in the Thinking Forth, Chapter 7, Section "Saving and Restoring a State" [Bro84]; in particular, this section cites Charles Moore as follows:

> You really get tied up in a knot. You're creating problems for yourself. If I want a hex dump I say `HEX DUMP`. If I want a decimal dump I say `DECIMAL DUMP`. I don't give `DUMP` the privilege of messing around with my environment.
>
> There's a philosophical choice between restoring a situation when you finish and establishing the situation when you start. For a long time I felt you should restore the situation when you're finished. And I would try to do that consistently everywhere. But it's hard to define "everywhere." So now I tend to establish the state before I start.
>
> If I have a word which cares where things are, it had better set them. If somebody else changes them, they don't have to worry about resetting them.
>
> There are more exits than there are entrances.

[*] Correspondence Address: Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria; `anton@mips.complang.tuwien.ac.at`

Unfortunately, this workaround is not even usable in our running example: What is the situation that should be established before the call to `.` in `foo`?

And this workaround does not help at all with other cleanup tasks like closing files and freeing other resources.

3.2 Using `catch`

Fortunately, with the introduction of `catch` in Forth-94, the situation changes: There is only one exit from `catch`, and we can use that property to make the cleanup more reliable:

```
: hex.-helper ( u -- )
  hex u. ;

: hex. ( u -- )
  base @ >r
  ['] hex.-helper catch
  r> base ! throw ;
```

This makes sure that `base` will be restored even if an exception (of any kind) happens while `hex.-helper` is executed.

Unfortunately, there is still one chink in our cleanup armour: If an exception (e.g., a user interrupt) happens during the restoration of `base`, the cleanup code would not complete, and `base` would be left in the wrong state.

4 Advanced solutions

4.1 `Try...restore...endtry`

The current development version of Gforth offers a construct `try` *`code1`* `restore` *`code2`* `endtry`. If any exception happens anywhere between `try` and `endtry` (including in *`code2`*), the stack depths are reset to the depth at `try`, the throw value is pushed on the data stack, and execution jumps right behind the `restore`.

With this construct there is not just only one exit, it also guarantees that *`code2`* is executed from start to end.

This construct can be used to solve our problem as follows:

```
: hex. ( u -- )
  base @ { oldbase }
  try
    hex .
    0       \ value for throw
  restore
    oldbase base !
  endtry
  throw ;
```

The old base is stored in a local, because we cannot use the return stack for this purpose (`try` pushes an exception frame on the return stack). However, Instead of using a local, we could use the data stack, as follows:

```
: hex. ( u -- )
  base @
  try
    over hex .
    0       \ value for throw
  restore
    over base !
  endtry
  throw
  2drop ;
```

Note how we use `over` twice to keep the values on the data stack intact, so we can use them in the restoration code. We only drop these values after `endtry`.

This construct requires some care in usage:

- As shown above, one has to be careful not to remove items from the stacks that are needed in the restoration; one must not even remove them during restoration.
- The restoration code must not throw an exception, at least not every time it executes. Otherwise the system will go into an infinite loop of start-restoration...throw-exception. And not even a user interrupt can be used to break out of that loop. Instead, the user then has to stop the system by using some more brutal methods (e.g., in Unix by sending a `SIGTERM` to the Gforth process).
- The restoration code must be idempotent, i.e., executing it multiple times (starting at the same stack depths) should have the same effect as executing it once.

However, Forth programmers are used to taking responsibility for their programs, so these caveats should not be a problem.

The idempotence requirement may be hard or impossible to satisfy in some cases, e.g., when the cleanup involves `close-file` or `free`. In such cases it is usually preferable to have a small chance of not cleaning up than to try to clean up several times. One can achieve this by writing the non-idempotent part between the `endtry` and the `throw`.

In cases where a variable is changed and restored, the idempotence requirement is easy to achieve.

An example of a non-idempotent use is:

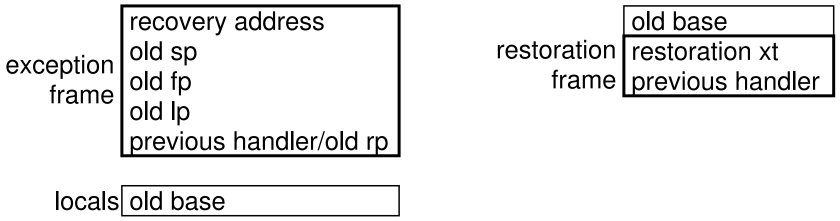


Figure 1: A general-purpose exception frame used for restoring `base` compared to a restoration frame

```
... open-file throw { f }
try
  ... f read-file throw ...
0 restore
endtry
f close-file throw
throw
```

4.2 Special-purpose words

Gforth also has special-purpose words for a few frequent or dangerous purposes:

`base-execute ( i*x xt u -- j*x )` executes *xt* while `base` is set to *u*.

`infile-execute ( i*x xt file-id -- j*x )` executes *xt* while `key` etc. read their input from *file-id*.

`outfile-execute ( i*x xt file-id -- j*x )` executes *xt* while the output of `type` etc. is redirected to *file-id*.

Given that all these words take an xt from the stack, and the xt is nearly always a constant, it is probably better to define future words of this kind such that they take the xt from the top-of-stack.

5 Efficiency

An exception frame costs five return stack cells in Gforth (and probably a similar amount in other systems), and constructing and consuming it costs a bit of time. For the purpose of cleanup a full exception frame is overkill. We don't really need to restore the depths of all stacks in this case: If we enter the restoration in the normal way, the stack depths are not restored anyway; and if we enter the restoration code through a `throw`, we are going to throw the error further on, so we don't need the stack depths, either; we just need access to the restoration data.

So we could implement a lighter-weight mechanism for restoration. Two cells for the restoration frame itself would be enough (see Fig. 1). The restoration frames would be kept on the return stack and chained in a linked list. `Throw` would process all the restoration frames that are above the next exception frame on the return stack, then process that exception frame as usual.

The downside is that the code for the restoration, and for setting up the restoration data would have to be even more aware of the restoration mechanism, because the restoration data cannot directly be transferred through a stack, but has to be accessed through the restoration frame. E.g., the restoration word for restoring `base` might look as follows:

```
: restore-base ( addr -- )
  dup @ base !
  next-restoration ;
```

Here, ***addr*** is the address of the user part of the restoration frame. `Next-restoration ( addr -- )` removes the current restoration frame from the chain. Any non-idempotent cleanup code would happen after `next-restoration`.

An implementation of `base-execute` with such a mechanism might look as follows:

```
: base-execute ( i*x xt u -- j*x )
  base @ >r
  ['] restore-base >restore
    base ! execute
  restore>
  r> drop ;
```

Here `>restore` would push a restoration frame on the return stack and add it to the restoration chain. `Restore>` would execute the restoration xt (i.e., `restore-base`) and drop it from the return stack. The old base would have to be dropped explicitly.

This mechanism has not been implemented. While it would be relatively easy to implement, it is unclear if it is worth the documentation and support load to provide it as a feature to the users. Here are a number of points to consider:

- In my experience nearly all uses of `catch` are for restoration/cleanup. So most exception frames could be replaced by lighter-weight restoration frames.

- Exception frames and their handling have not shown up as performance bottlenecks, but then I have not performed any measurements.

6 Related work

I am not aware of other advanced solutions in Forth. However, this is a common programming problem, so other languages have developed a wide variety of approaches for solving it.

6.1 Dynamic Scoping

Some of the problems addressed in this paper, e.g., our `base`-based running examples can be seen as customizing the execution environment. Hanson and Proebsting [HP01] argue that dynamically-scoped variables have the right properties for this usage and that programmers in languages without dynamic scoping resort to simulating dynamic scoping, and they point out the similiarity between exceptions (a dynamically scoped control structure) and dynamically scoped variables (which explains why we and others use exception-catching to implement them).

A significant number of programming languages and systems provide dynamically-scoped variables. Lisp is a well-known example. But a probably more widely-used example is environment variables in Unix and Windows processes.

Another language with dynamically-scoped variables is Postscript; there programmers perform dynamic scoping by (in Forth terminology) constructing wordlists dynamically, and pushing them on the search order stack; because name lookup in Postscript happens at run-time, this results in dynamic scoping. However, the Postscript dynamic control-flow words (`exit`, `stop`) do not affect the depth of the control-flow stack, so these features cannot be combined safely.

6.2 Cleanup

Lisp has the `unwind-protect` special form[1]: `(unwind-protect` *`protected cleanup`*`)` makes sure that *`cleanup`* is executed in any case, even if there is an abnormal exit from *`protected`*. However, unlike `try...restore...endtry`, it does not protect against abnormal exits from *`cleanup`*.

Java has a similar feature in the form of the `try ... finally` construct, and C++ in `try ... catch`.

C++ also provides destructors that can be used to automatically release resources and perform other cleanup when the scope of a variable is exited. Stroustroup[Str01] gives a good overview of what kind of exception safety are desirable, and how the various features of C++ may be used to achieve them.

In a similar vein, Java finalizers perform cleanup actions when an object is garbage-collected. However, because the finalizer may be executed a long time after a destructor would have been executed, it is often recommended to favor other approaches over using finalizers.

Many other languages have similar features.

7 Conclusion

The introduction of `catch` in Forth-94 provided a good basis for writing code that cleans up after itself rather than requiring every piece of code to clean up all the trash that all other code may have left behind.

However, in the presence of user interrupts and other asynchronous exceptions this is not sufficient. We propose the `try ... restore ... endtry` construct that can be used to solve this problem completely for some, but not all uses. We also discuss a more light-weight implementation technique.

References

[Bro84] Leo Brodie. *Thinking Forth.* Fig Leaf Press (Forth Interest Group), 100 Dolores St, Suite 183, Carmel, CA 93923, USA, 1984.

[HP01] David. R. Hanson and Todd A. Proebsting. Dynamic variables. In *SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 264–273, 2001.

[Str01] Bjarne Stroustrup. Exception safety: concepts and techniques. In *Advances in exception handling techniques*, pages 60–76. Springer LNCS 2022, 2001.

[1] `http://www.lispworks.com/documentation/HyperSpec/Body/s_unwind.htm`

Forth System Hooks — Metaobject Protocol in Forth Style

Ulrich Hoffmann
uh@fh-wedel.de

September 18, 2008

Abstract

The ANS Forth Standard and its follow up standard effort, Forth 200x, allow for writing Forth applications portable to a wide range of commercial and open source Forth systems. One area of Forth which has been notoriously hard to standardize is the area of meta/target compilers as well as advanced compiler and interpreter extensions.

Many Forth systems implement system specific extension mechanisms in order to support just their special meta/target compiler but there is no common practice how to do so. One extension mechanism is to place execution vectors — or *hooks* — at key positions in the Forth system. In the un–extended case, the vectors have no or a default behavior and in the extended case the can get sophisticated and elaborated behavior.

One example of a very simple hook would be a vector *notfound* in the typical outer interpreter, which would be located right at the end when both, token lookup in the dictionary and the conversion of the token to a number, failed. In the un–extended case *notfound* would issue an error message (*complaint*) about an unknown token. In the extended case, say hexadecimal number input with `$`–prefix, *notfound* could try a hexadecimal number conversion, and leave the appropriate number on the stack, or issue an error message as before.

We could write a hexadecimal number input with `$`–prefix as a portable Forth system extension, if we would agree on the hook `notfound` and its default behavior.

Other extensions that come to mind are object oriented systems with advanced search order requirements such as Manfred Mahlows CSP (Context Switching Prelude) system. It also can benefit from well defined system hooks.

The idea of hooks is quite common, not only with Forth system but also with operating systems in general. The EMACS editor provides a vast collection of hooks for all kinds of extensions.

In the context of Common Lisp's Object System CLOS, a technique called the Meta Object Protocol has been developed by Gregor Kiczales in the early 1990s which allowed to implement an early form of aspect oriented programming. The advantage of their unique approach was that the CLOS community widely accepted the functionality of their hook pendants as defined in the book *The Art of Meta Object Programming* (AMOP). As a result AMOP based CLOS extensions were portable over a wide variety of CLOS implementations.

The talk will give a short overview of hooks and the CLOS meta object protocol and will also propose possible Forth system hooks in traditional Forth implementations with hope to come closer to a Forth system hook standardization.

Virtual Machine Showdown: stack versus registers

Yunhe Shi[1], Kevin Casey[1], Anton Ertl[2], **David Gregg**[1]

[1]Department of Computer Science
Trinity College Dublin

[2]Institut für Computersprachen
Technische Universität Wien

This talk is based on a paper in *ACM TACO* 4(4), January 2008

Virtual Machines (VM)

- High-level language VMs
  - Popular for implementing programming languages
    - *Java, C#, Pascal, Perl*
- Program is compiled to virtual machine code
  - Similar to real machine code
  - But architecture neutral
- VM implemented on all target architectures
  - Using interpreter and/or JIT compiler
  - Same VM code then runs on all machines

Stack Architecture

- Almost all real computers use a register architecture
  - Values loaded to registers
  - Operated on in registers
- But most popular VMs use stack architecture
  - Java VM, .NET VM, Pascal P-code, Perl 5

Why stack VMs?

- Code density
  - No need to specify register numbers
- Easy to generate stack code
  - No register allocation
- No assumptions about number of registers
  - ????
- Speed
  - May be easier to JIT compile
  - May be faster to interpret
    - Or maybe not…

VM Interpreters

- Emulate a virtual instruction set
- Track state of virtual machine
  - Virtual instruction pointer (IP)
  - Virtual stack
    - Array in memory
    - With virtual stack pointer (SP)
  - Virtual registers
    - Array in memory
    - No easy way to map virtual registers to real registers in an interpreter

VM Interpreters

```
while ( 1 ) {
  ip++;
  opcode = *ip;
  switch ( opcode ) {
    case IADD:        *(sp-1) = *sp + *(sp-1); sp--; break;
    case ISUB:        *(sp-1) = *sp – *(sp-1); sp--; break;
    case ILOAD_0:     *(sp+1) = locals[0]; sp++; break;
    case ISTORE_0:    locals[0] = *sp; sp--; break;
    .........
  }
}
```

Which VM interpreter is faster?

- Stack VM interpreters
  - Operands are located on stack
  - No need to specify location of operands
  - No need to load operand locations
- Register VM interpreters
  - Fewer VM instructions needed
    - Less shuffling of data onto/off stack
  - Each VM instruction is more expensive

Which VM interpreter is faster?

- Question debated repeatedly over the years
  - Many arguments, small examples
  - No hard numbers
- Some are confident that answer is obvious
  - But which answer?

VM Interpreters

- Dispatch
  - Fetch opcode & jump to implementation
  - Most expensive part of execution
  - Unpredictable indirect branch
  - Similar cost for both VM types
  - But register VM needs fewer dispatches
- Fetch operands
  - Locations are explicit in stack machine
- Perform the operation
  - Often cheapest part of execution

Operand Access

- Stack machine
  - Virtual stack in array
  - Operands on top of stack
  - Stack pointer updates
- Register machine
  - Virtual registers in array
  - Must fetch operand locations (1-3 extra bytes)
    - More loads per VM instruction

Stack versus registers

- Our register VM
  - Simple translation from JVM bytecode
  - One byte register numbers

Source code	**Stack code**	**Register code**

a = b + c;	iload b;
	iload c;
	iadd;
	istore a;
		iadd a, b, c

From Stack to Register

- Translated JVM code to register VM
- Local variables mapped directly
  - Local 0 → Register 0
- Stack locations
  - Mapped to virtual registers
  - Height of stack is always known statically
  - Assign numbers to stack locations

From Stack to Register

Stack Code	Register Code	Comment
iload 4	imove r10, r4	; load local variable 4
bipush 57	biload r11, 57	; push immediate 57
iadd	iadd r10, r10, r11	; integer add
istore 6	imove r6, r10	; store TOS to local 6
iload 6	imove r10, r6	; load local variable 6
ifeq 7	ifeq r10, 7	; branch by 7 if TOS==0

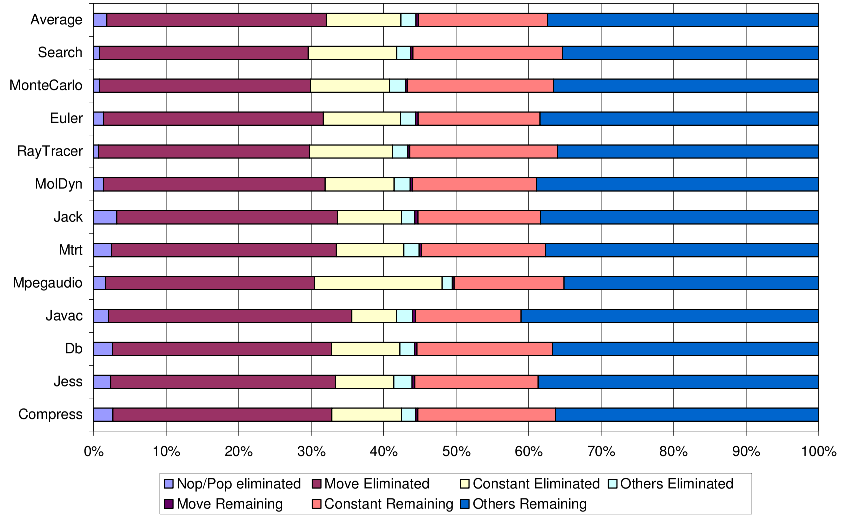
From Stack to Register

- Clean up register code with classical optimizations
  - Copy propagation to remove unnecessary move operations
  - Partial redundancy elimination
    - Re-use constants already in registers
    - Stack VM consumes its operands so must load constants every time it uses them

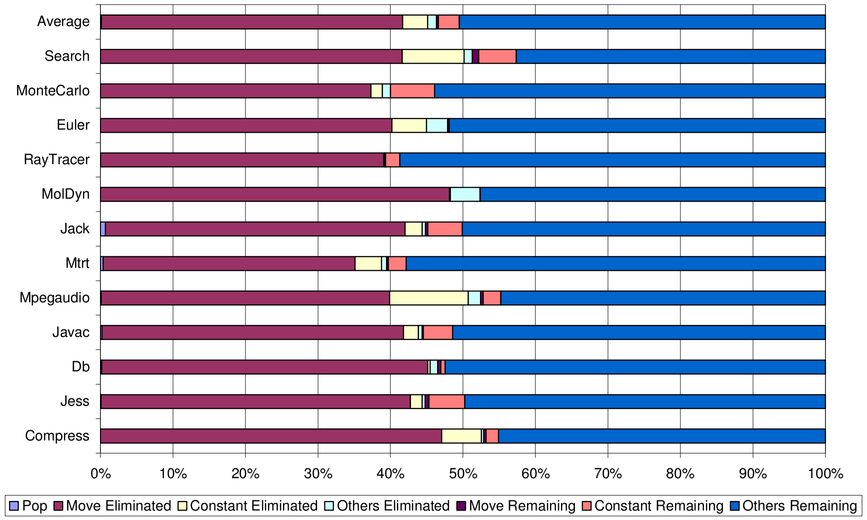
Experimental Setup

- Implemented in Cacao VM
- Method is JIT compiled to register code on first invocation
  - Results include only executed methods
- Standard benchmarks
  - SPECjvm98, Java Grande
- Real implementation wouldn't translate
  - Better generate register code from source
  - But translation allows fairer comparison
    - Except for translation time

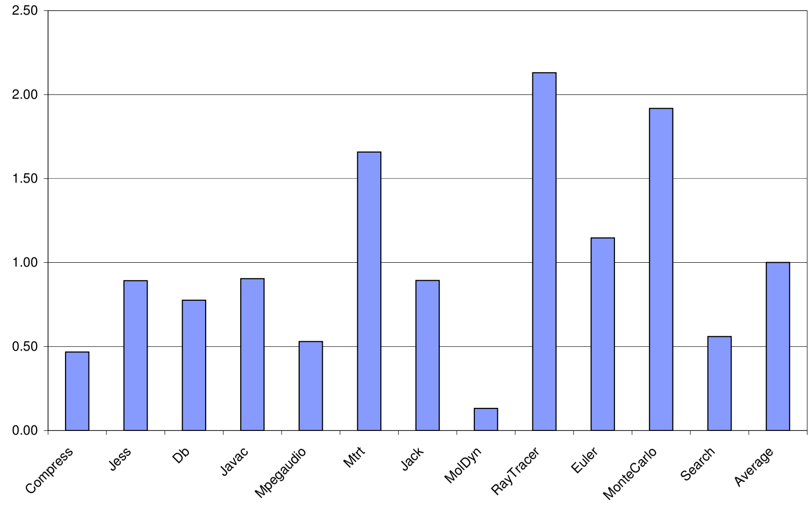
Static VM Instructions



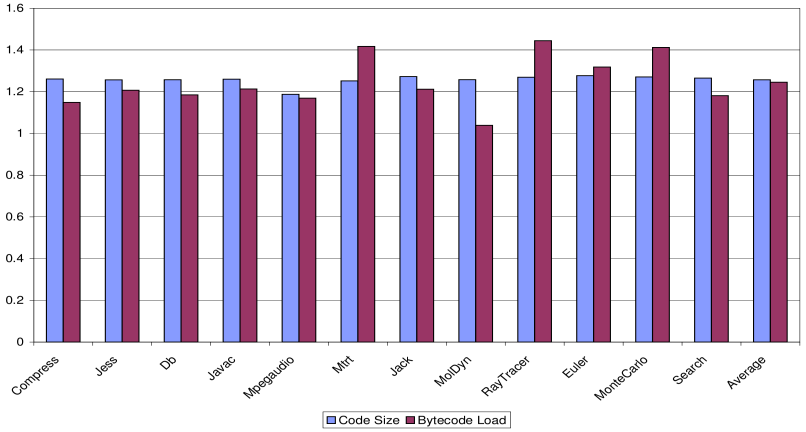
Dynamic VM Instructions



Ratio of additional loads to eliminated instructions



Increase in bytecode loads



Real machine memory ops

Source Code

```
a = b + c;
```

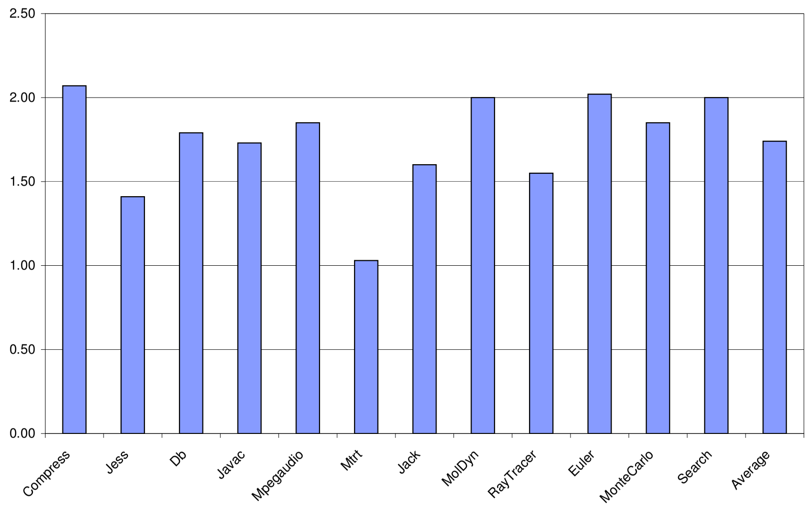
Register Code

```
/* iadd a, b, c */
reg[a] = reg[b] + reg[c];
```

Stack Code

```
/* iload c */
*(++sp) = locals[c];

/* iload b */
*(++sp) = locals[b];

/* iadd */
*(sp-1) = *(sp-1) + *sp;
sp--;

/* istore a */
locals[a] = *(sp--);
```

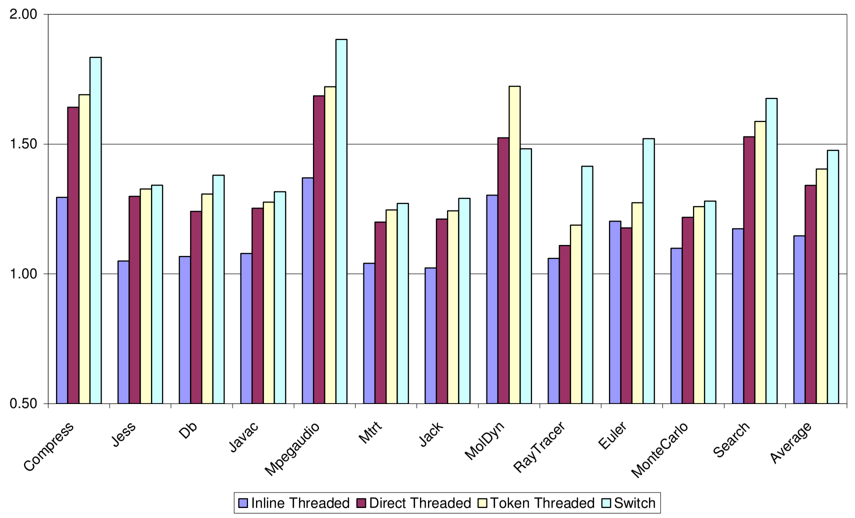
Reduction in “real machine” loads/stores compared with dispatches eliminated



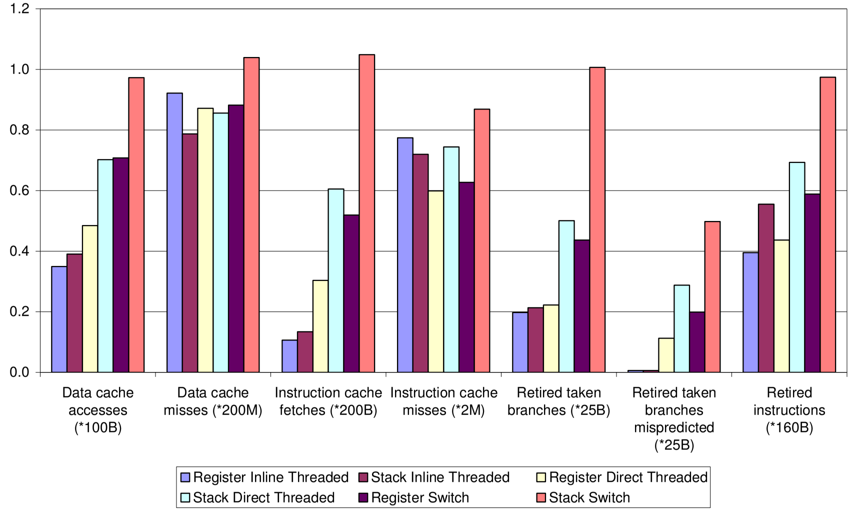
Real Running Times

- Interpreter Dispatch
  - Switch dispatch
  - Token Threaded dispatch
  - Direct threaded dispatch
  - Inline threaded dispatch
- Hardware platforms
  - AMD 64
  - Intel P4
  - Intel Core 2 Duo
  - Digital Alpha
  - IBM PowerPC

Speedup of Register VM – AMD64



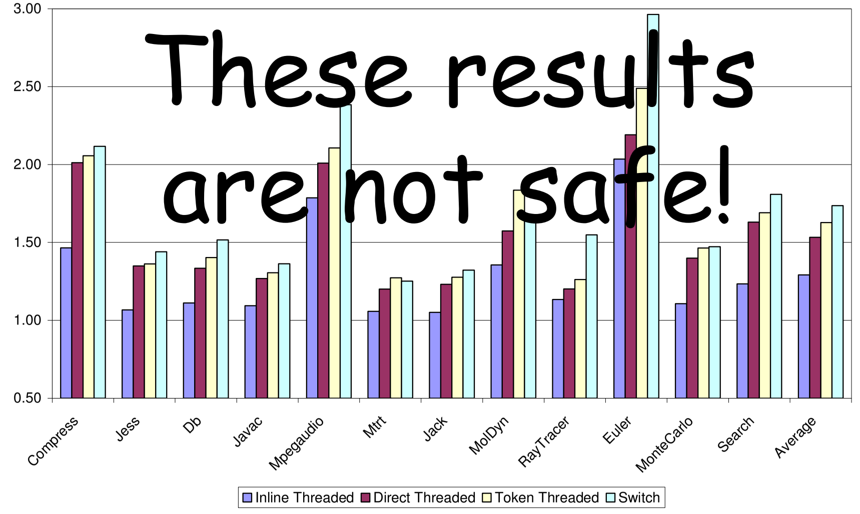
AMD64 Event Counters - Compress



Eliminating more redundant expressions

- Stack operations consume their operands
  - So very difficult to re-use existing values
  - Stack machine must load constants, loop invariants repeatedly
  - Register machine can store constants, simple loop invariants in registers
- What about more complex invariants
  - Repeated loads from the heap
  - Requires very sophisticated pointer analysis
    - But what if we could do it?

Eliminating more redundant expressions – speedup on AMD 64



These results are not safe!

Java VM Summary

- Detailed quantitative results
  - 46% reduction in executed VM instructions
  - 26% increase in bytecode size
  - 25% increase in bytecode loads
- Speedup depends on dispatch scheme
  - Speedup 1.48 with switch dispatch on AMD64
  - Even with the most efficient dispatch, 1.15 speedup can still be achieved

What about Forth?

- Forth usually uses stack VM
- But execution profile very different
- Java instructions:
  - 42% load & stores of locals
  - 6% loads of constants
  - 0-2% stack manipulation
- Very many local load/store
  - Almost all disappear in register VM

What about Forth?

- Forth VM instructions
  - Stack manipulation instructions
    - over, dup, swap, drop, 2dup, ?dup, r>, >r, i
    - maybe 10%-15% ???
  - Literal instructions
    - lit, var
    - maybe 15%-25% ???
  - Local variable instructions
    - >l, @local
    - maybe 2%-5% ???

What about Forth?

- There is no huge block of instructions that will easily disappear using a register VM
  - Apart from literals
- But some speedup is probably possible by using a register VM

An infix syntax for Forth (and its compiler)

Andrew Haley

Early Inspiration

- Winfield AFT, ‘Pascal in Forth’, SOFT, Vol 1, no 4, Sept. 1983, pp59-63 and Vol 1, no 5, Oct. 1983, pp46-51.
http://www.ias.uwe.ac.uk/~a-winfie/aw_publications.htm
- Very elegant, but closer to Pascal than to Forth – the resulting syntax is more restricted, and the control structures are those of Pacal, not Forth. Also, restricted to single-length integer expressions and arrays, no structures, etc, etc.

Previous efforts

- Forthwrite Dec ’86:

```
VARIABLE 'EXPRESSION  : EXPRESSION  'EXPRESSION @EXECUTE ;
VARIABLE TEMP  CREATE )
: ,C ( a)   2- , ;
: NEXT ( - a)   -' IF  NUMBER TEMP !  0  ELSE DROP  THEN ;
: CHECK ( a a')  - ABORT" not matched" ;

: FACTOR ( a - a')   DUP ['] ( = IF  DROP  NEXT  EXPRESSION
      ['] ) CHECK  ELSE  ?DUP IF  ,C
         ELSE  TEMP @ [COMPILE] LITERAL  THEN  THEN  NEXT ;
: TERM ( a - a')   FACTOR  BEGIN  DUP ['] * =  OVER ['] / =
      OR WHILE  NEXT FACTOR  SWAP  ,C  REPEAT ;
: EXPRESSION ( a - a')   TERM  BEGIN  DUP ['] + =  OVER ['] - =
      OR WHILE  NEXT TERM  SWAP ,C  REPEAT ;

: INFIX   NEXT ['] ( CHECK  NEXT EXPRESSION  ['] ) CHECK ;
  IMMEDIATE    ' EXPRESSION 'EXPRESSION !

Example of use:

  44 CONSTANT FRED
  : TEST ( -- n )  INFIX ( 3 * FRED / ( ( 3 + 5 ) / 2 ) ) ;
```

Previous efforts

- Forthwrite Dec ’86:
  - Uses recursive descent
  - Compile only – no use in interpreter
  - No LOCAL variables
  - Extremely simple
  - Only arithmetic expressions
  - Uses data stack
  - Uses `-'` (aka `FIND`) and `,C` (aka `COMPILE,`)

Previous efforts

- comp.lang.forth Feb 2002, some details elided:

```
: op ( a)   state @ if  compile,  else execute  then ;
: lit   =number @  state @ if  postpone literal  then ;

ops[ relop   > > < < = = ]
ops[ addop   + + - - or or xor xor ]
ops[ mulop   * * / / and and ]
ops[ unop   - negate @ @ ]

\ These are the productions.

defer expr
: expr-list   expr  begin  match ,  while  token expr  repeat ;

: parens  expr-list  match ) 0= abort" )" ;

: primary
   match# if  lit  token  exit  then
   match ( if  token parens  token  exit  then
   this >r  token  match ( if  token  parens  token  then  r> op ;

: factor   unop if  >r  token  recurse  r> op  exit  then  primary ;

: term   factor
   begin  mulop while  >r  token  factor  r> op  repeat ;

: simple-expr   term
   begin  addop while  >r  token  term  r> op  repeat ;

:noname   simple-expr
   begin  relop while  >r  token  simple-expr  r> op  repeat ;
is expr
```

Previous efforts

- comp.lang.forth Feb 2002:
  - Uses recursive descent
  - STATE-smart: allows interpretive use
  - Still extremely simple
  - Function calls: `FOO ( 1, BAR, 3 )`
  - Uses return stack for temporary storage of execution tokens that haven't yet been used because they are of low precedence – much cleaner; means we can use data stack for interpretive expression evaluation
  - Written in almost Standard Forth
  - Still doesn't allow LOCAL variables in expressions

The problem with locals

"Words that return execution tokens, such as `'` (tick), `[']`, or `FIND`, shall not be used with local names."

This is a horrible restriction! Effectively it means that locals can never be used as factors. Locals cannot be used as part of an expression in this parser because it uses `'` and `COMPILE,`

Designing the syntax

- Let's ignore the implementation problems for a little while and look at the syntax we'd like to have. We'll return to the implementation later.

Designing the syntax

- A word is any string of non-whitespace characters. Words are separated by spaces.
- Numbers are just words, so they don't need to be treated specially. The syntax need make no special provision for them.

Designing the syntax

- Simple cases:
  - Basic Forth syntax is

    `noun noun ... verb  noun noun ... verb`

  profanely,

    verb (noun , noun , ...) ; verb (noun , noun , ...) ;
- Control structures:
  - `a b > if` becomes if (a > b)
  - `10 0 do` becomes do (10 , 0)

Designing the syntax

- More simple cases:
  - Arithmetic expressions:
    - Traditional operator precedence, defined by syntax

      `b negate b b * 4 a * c * - sqrt 2 / a * +`

      becomes

      - b + (sqrt (b * b - 4 * a * c) / 2 * a)

  The reserved tokens are

    + - * / f+ f- f* f/ ( ) < > = f< f> f= or xor and @

  Everything else is just a word, and can be used as a function or an argument.

Designing the syntax

- To allow multiple statements, we add the ; operator:

  expr ; expr
- Local variables can be assigned with the := operator:

  `a b * to c` becomes c := a * b
- @ is a problem. We could just treat it as a function like any other Forth word, but then it would be cumbersome to use because of parentheses:

  @ (a) + @ (b) ...

  so we define @ to be a high-precedence unary operator, which is much nicer:

  @ a + @ b ...
- We could arguably do the same with `!` , treating it as a binary operator

Designing the syntax

- A structure access, as per the Forth 200x structures RFD, is just the application of a function to a pointer.
  Given a struct, we can use its fields with no special treatment:

```
struct point
    1 cells +field p.x
    1 cells +field p.y
end-struct
\ Draw a line from p1 to p2
draw ( p.x ( p1 ) , p.y ( p1 ) , p.x ( p2 ) , p.y ( p2 ) ;
```

- We could define a word `.` as a postfix function operator, but that isn't obviously a big improvement

Designing the syntax

- Because every statement is also an expression, we can have conditionals in expressions, so:

```
    a := b + ( if ( c < 10 ) ; 1 ; else ; 2 )
```

  is equivalent to

```
    b c 10 < if 1 else 2 + to a
```

Designing the syntax

- I'm still not certain about the absolute best syntax for arrays, but Smalltalk is a good place to start
- For array reads,
  - a at: i produces `a i at:`
- And for writes,
  - <expression> put: (b , 2) produces `b 2 put:`
  - (Maybe b at: 2 put: <expression> would be better)
- With an additional shorthand (purely for familiarity's sake):
  - a [i] is equivalent to a at: i

Designing the syntax

- Arrays are tricky. In profane languages *lvalues* are treated differently from *rvalues*: an lvalue is evaluated for its address, but an rvalue is evaluated for its value
- For example,

```
a [ i ] := b [ j ]
```

- We can't simply say that every array access on the LHS of an assignment is evaluated for its address, because of things like

```
a [ b [ i ] ] := b [ j ]
```

  where only the *outermost* array access is evaluated for its address
- It's difficult to do a mapping in a purely syntactical way. If we're simply scanning from left to right we have no way to know that an assignment is imminent; that would require *backtracking*

Designing the syntax

- Parsing words are the biggest headache. Anything that acts as a prefix operator by using `PARSE` or `WORD` needs special treatment
- String constants are easy enough, though:

```
    s" hello " type
```

  maps easily to

```
    type ( " hello " )
```

- I don't think the lack of `."` is important

Escape to Forth

- If all else fails and there really is a Forth expression that cannot be rendered as infix in any way, there's an escape:

```
[." Hello, world"]
```

- This also allows local declarations, etc:

```
[ LOCALS| a b c |]
```

The problem with TO

"An ambiguous condition exists if either `POSTPONE` or `[COMPILE]` is applied to `TO`."

So `TO` can never be used as a factor either.

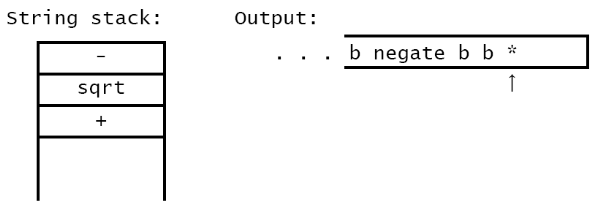
This is a very bad design decision: if Forth is about any single thing it's factoring, and this is an important part of the language that *forbids* factoring.

Implementation

- The problem with TO not being allowed to be ticked or POSTPONED was, as it turned out, a big inspiration
- We can't use XTs, but we can use strings. So, instead of saving XTs on the return stack, we create a string stack and define `>S` and `S>`. Also, we create an output buffer and push into it words from the string stack
- At any stage in the compilation, we only have to decide whether to push a word into the output buffer or onto the string stack

Implementation

```
Source:
-b + sqrt ( b * b - 4 * a * c )
                  ↑
                 >IN
```



Implementation

A great benefit – arguably *the* greatest benefit – of doing this by using strings rather than XTs is that we no longer need to be `STATE`-smart. The infix code is rewritten to be postfix and then passed to `INTERPRET`. `INTERPRET` either compiles or interprets.

An example

Original FORTRAN:

```
do i = 1, dim1
   do j = 1, dim3
     C(i, j) = 0.
     do k = 1, dim2
        C(i, j) = C(i, j) + A(i, k)*B(k, j)
     enddo
   enddo
enddo
```

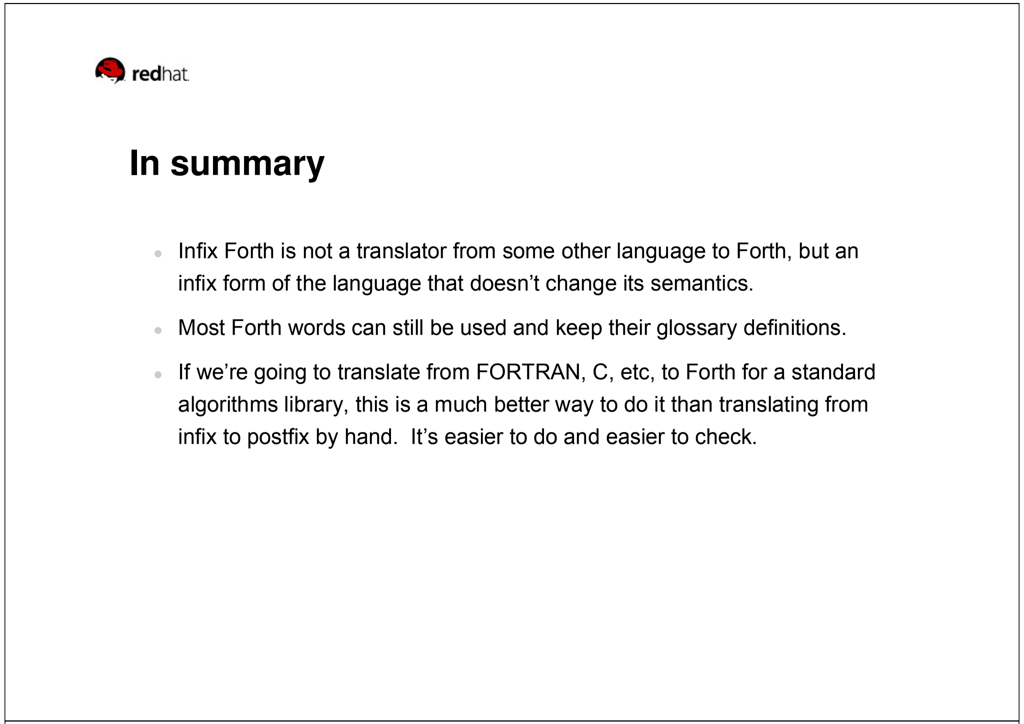
An example

Infix Forth:

```
do ( dim1 , 1 ) ;
   do ( dim3 , 1 ) ;
     0.e0 put: C ( j , i ) ;
     do ( dim2 , 1 ) ;
        C [ k , j ] f+ A [ k , i ] f* B [ i , j ] put: C ( k , j ) ;
     loop ;
   loop ;
Loop
```

generates

```
dim1 1 do
  dim3 1 do
    0.e0 j i C put:
    dim2 1 do
      C k j at: A k i at: B i j at: * + k j C put:
    loop
  loop
loop
```



In summary

- Infix Forth is not a translator from some other language to Forth, but an infix form of the language that doesn't change its semantics.
- Most Forth words can still be used and keep their glossary definitions.
- If we're going to translate from FORTRAN, C, etc, to Forth for a standard algorithms library, this is a much better way to do it than translating from infix to postfix by hand. It's easier to do and easier to check.

Filters for Unicode Markup, a work in progress report.

Bill Stoddart and Angel Robert Lynas
University of Teesside, UK

September 19, 2008

Abstract

The advent of Unicode extends the restrictive ASCII character set with millions of characters. However, we still have the same keyboards! Very often a document or program will require a limited set of Unicode characters in addition to characters available from the keyboard. We describe an approach which allows a user to provide a "markup" for each special character required, such as `\alpha` for the Greek letter α, and in which the markup is replaced by the character it represents when typed at the keyboard or streamed from file. The techniques used include vectoring Forth's keyboard input to enable markup sequences to take effect during Forth command line input, and writing Unix filters in Forth. The latter allows the provision of wrappers that allow the use of Unicode markup with any Unix editor that can accept input from the standard input device.

keywords. Forth, Unicode, Unix Filters, UTF-8, RVM-Forth

1 Introduction

The Unicode Standard, available at `unicode.org`, provides a description of an extended character set encompassing mathematical symbols and the alphabets of most natural languages. The matter of character encodings is separated from the form of the characters themselves, and the encoding that has become dominant within Linux distributions is known as UTF-8, in which characters are represented within files or computer memory as a sequence of between one and four bytes. Support for Unicode is now becoming widely available, although it is still not included in some well known packages, for example Lesstif, the freeware version of Motif. Incorporation of a UTF-8 locale is non-trivial, as it breaks an implicit assumption under which almost all packages have been written, that each character occupies an equal amount of space. The Forth 93 Standard looked forward towards internationalisation, but assumed characters would remain of equal length but might become longer. Whilst such encodings are defined in Unicode, they have not proved to be the most popular ones, and anyone wishing to display a Unicode character on an Linux X Windows terminal or editing window will probably need to do it using the variable length encodings of UTF-8.

Unicode encoding issues were raised in 1998-9 by Stephen Pelc, Steve Coul and Peter Knaggs. An updated discussion from Stephen Pelc and Peter Knaggs appeared in 2001. An RFD on extended characters, designed to cope with all Unicode encodings, has been posted to the Forth Standards forum on forth200x.org by Bernd Paysan. A discussion paper “Xchars, or Unicode in Forth” by Anton Ertl and Bend Paysan appeared in EuroForth 2005 proceedings. Support for wide characters has been included in GForth and BigForth.

We are currently including UTF-8 support within our RVM-Forth (Reversible Virtual Machine Forth) environment, to gain access to mathematical symbols and other symbols commonly used in mathematics, such as letters from the Greek alphabet. We do this by providing a markup sequence for each special character required, e.g. `\alpha` for the Greek letter α. When the markup sequence is entered from the keyboard, it is immediately replaced by the Unicode character it represents. We provide this facility at the Forth command line, and also make it available for use in editing via a Unix filter, also written in Forth.

2 Preparing RVM-Forth to act as a Unix Filter

RVM-Forth is a subroutine threaded Forth. It has a small nucleus generated by “meta compilation” of a description of Forth written in Forth. Metacompilation of the nucleus generates a Gnu Assembler file which is linked with object code generated from parts of the nucleus which are written in C. This provides an executable Forth Nucleus. When the Forth Nucleus is invoked, command line arguments are provided in the usual C style, and these are converted to a single string which is interpreted as though it where Forth terminal input. The usual startup sequence is:

```
RVM-FORTH HI
```

Where HI is a command that loads additional utilities, provided as Forth source code files, which are incrementally compiled. This process appears to the user to take virtually no time.

To use RVM-Forth to provide a UTF-8 markup filter when editing the file example.r with the Nano editor, we invoke it from a script file NANO as:

```
NANO example.r
```

Where NANO contains:

```
#! /usr/bash
RVM_FORTH UTILS ALSO UTF8 FILTER | nano -c -O -E -T 3 $1
stty echo icanon
```

Here the command UTILS loads the same utilities as HI, but does not print a sign on message. Forth is taking its input from the keyboard, and piping its output to Nano.

Unix, by default, uses buffered keyboard input in which a whole line is input and echoed to the terminal before the characters are seen by a program. RVM-Forth configures Unix keyboard input to be unbuffered, so that it receives one character at a time without that character being echoed to the screen. The second line of the file is included to restore normal Unix terminal handling when the filter terminates.

The commands ALSO UTF-8 FILTER add the Forth wordlist UTF-8 to Forth's search order, then invokes the command FILTER, which is from this wordlist. FILTER records keyboard input in a circular "markup buffer", as well as generally passing it through to standard output. Whenever a defined markup sequence matches with the characters most recently received into the markup buffer, FILTER reinitialises the markup buffer, outputs characters to delete the markup sequence from the edit screen, and replaces it with the associated Unicode character.

The Forth side of the filter is non-terminating, and overall termination occurs when the user exits from the editor.

A second possible use of the same filter is as

```
RVM_FORTH UTILS ALSO UTF8 FILTER < raw.r > cooked.r
```

In this case Forth is receiving its input from file and must leave configuration of terminal input to the Unix filter mechanism. To achieve this Forth needs to test whether its input is coming from a keyboard during its configuration sequence, which it does by issuing a Unix system call via the C function:

```
int stdin_is_kbd() /*return -1 (Forth true flag) if stdin is the
keyboard, and 0 if stdin has been redirected to a pipe or file */
{ if ( system("[ -t 0 ]") == 0 ) return -1; else return 0 ;
}
```

The functionality of this command is made available as a Forth command by the following code[1]

```
CODE STDIN_IS_KBD ( -- f, returns true if stdin is the keyboard )
  xchg %esp,%esi
  call stdin_is_kbd
  push %eax
  xchg %esp,%esi
  ret
ENDCODE
```

A second issue is that, from our observations of Unix filter behaviour, it seems that FILTER will need to terminate when the end of the input file is reached. The different termination behaviours required of FILTER are provided by the loop test in the following definition.

```
: FILTER ( --, pre: stdin is the keyboard or a file.
Filter input, replacing markup sequences with the
corresponding UTF-8 character codes )
  INITIALISE-BUFFER
  BEGIN ?KEY STDIN_IS_KBD OR WHILE FILTER-KEY REPEAT BYE ;
```

When receiving from a file, FILTER will terminate when no further bytes are available, which occurs at he end of the file. At this point (and not before) ?KEY returns a false flag.

Summarising, to enable RVM-Forth for Unix filter applications we have had to control the output of the RVM-Forth sign on message to ensure it does not occur during Filter deployment, and make standard input configuration dependant on whether the system finds it is receiving input from a keyboard or a file. When running RVM-Forth as a filter which pipes its output to an editor, the Forth filter need not explicitly terminate as it will be terminated externally when the editor terminates. When Forth runs as a filter which receives its input from a file, it must terminate when the end of the file is reached, and this will correspond to ?KEY returning a false flag.

[1] We do not use the elegant postfix syntax of the classical Forth assemblers because our assembly code is passed through to the Gnu assembler after processing for control structures and code definitions, making it convenient to use the Gnu AT&T syntax for our assembler commands.

3 Recognising and acting upon markup sequences

The association between markup sequences and UTF-8 character encodings is recorded as a sequence of ordered pairs using the RVM-Forth Sets Package.

Part of the sequence which records Greek alphabetic characters, for example, is:

```
STRING INT PROD [ " \GAMMA " CE93 |-> ,  " \DELTA " CE94 |-> ,
" \THETA " CE98 |-> ,  " \LAMBDA " CE9B |-> , " \XI " CE9E |-> ,
" \PI " CEA0 |-> ,  " \SIGMA " CEA3 |-> ,  " \PHI " CEA6 |-> ,
.. ]
```

Here we are constructing a sequence of string integer pairs. The open square bracket is a start sequence bracket. Strings are enclosed by quotes, and |-> is the maplet symbol, which removes two elements from the stack and constructs an ordered pair. The following comma compiles the ordered pair as the next sequence element.

"\GAMMA " is the markup string for a capital gamma (Γ) character, and CE93 is the UTF-8 hexadecimal encoding for the character, which occupies two bytes. As each keyboard character is entered, it is added to the circular markup buffer. The text in the markup buffer is then compared with the markup sequences to see if any markup sequence matches the most recent text in the buffer. If it does, we re-initialise the markup buffer, backspace and erase the markup sequence on the screen, and output the UTF-8 character.

Some input keys require special treatment. A backspace results in the last character in the markup buffer being removed if the buffer is not empty. A new line will clear the markup buffer.

No attempt is made to handle escape sequences. These are recorded in the markup buffer and passed though to standard output like normal key strokes. This means, for example, that using the cursor movement keys during entry of a markup will prevent that markup being recognised. This can actually be useful, as it provides a way of entering a markup sequence without having it transformed into its corresponding Unicode character. It also means, however, that spurious markups can occur. For example, the left arrow key generates the hex byte sequence 1B 5B 44. Since 5B 44 corresponds to [D, then if we include [D as a markup sequence, it will be spuriously recognised when a left arrow key is entered.

4 Experience with Unix editors

We have previously integrated Nedit into our Forth IDE. Typing `SEE <word>` at the Forth command prompt will cause Nedit to open the source file containing the definition of `<word>` in read only mode at the line where it is defined. `EDIT <word>` will open the file at the same place but in read/write mode. Nedit has the advantage of supporting a read only mode and of displaying line numbers. Unfortunately it is not suitable for UTF-8 encodings as it is based on Lesstif, which currently has no UTF-8 locale support.

Gedit handle files with extended characters, and some old versions of Gedit claim to have options for accepting input from STDIN. This option is not recorded on current documentation however, and we have not been able to make Gedit work with our UTF-8 filter. Another disadvantage is that there is no explicit read only mode. We have nevertheless implemented an option for Gedit to be the associated editor, and we use a Bash script to provide a read only wrapper for Gedit when a browsing mode is required.

Kate can also handle extended characters, and looked a promising candidate for use with a filter. In the Kate handbook,

(docs.kde.org/stable/en/kdesdk/kate/kate.pdf)

we read, under command line options:

```
kate --stdin
 Reads the document content from STDIN. This is similar to the
 common option - used in many command line programs, and allows
 you to pipe command output into Kate.
```

Whatever this might mean, it does not allow a filter configuration similar to the one used with Nano above. Standard output can be piped into Kate with commands such as:

```
 ls | kate --stdin
```

but we have not been able to use Kate with our UTF-8 filter in such a way that user input to Kate is passed through the markup filter.

Nano needs no special option to accept input from STDIN, and it works perfectly with the filter when the script NANO is invoked from the Unix Shell. However, when Nano is invoked from a system command sent by Forth, as when using `EDIT`, it sometimes fails to hit the correct line in the source file and also does not always record new lines correctly during the subsequent editing session.

5 Markup filtering of the Forth command line

The markup of utf-8 characters is achieved at the Forth command prompt by vectoring the execution of the Forth Standard `ACCEPT` to `UTF8-ACCEPT`. `ACCEPT` accepts the input of a given number of characters from the console into an input buffer, displaying characters as they are entered and terminating on receiving a new line or obtaining the given number of characters. `UTF8-ACCEPT` has the additional behaviour of storing received characters in the markup buffer and checking the markup buffer for a match against the possible markup sequences, such as `\alpha` . When a match is found, backspaces are output to delete the markup sequence from the screen, and the corresponding Unicode character is output in its place. Termination occurs on receipt of a new line character, or when the given maximum number of bytes have been received into the input buffer. Input of a backspace character causes the last character in the markup buffer to be removed, if it exists, and causes the last character of the input buffer to be removed. This latter may occupy between one and four bytes.

6 Adapting RVM-Forth to UTF-8 Unicode

Switching character representation to UTF-8 revokes the assumption under which RVM-Forth was written, that each character occupies one byte, and also the assumption, taken in the Forth Standard, that each character occupies an equal amount of memory space known as a "char location". With UTF-8 encoding a string of n characters may occupy more than n address units, leading to possible buffer overruns if we continue to rely on our old assumptions.

We use the abbreviation *pchar*, introduced Stephen Pelc and Peter Knaggs. A *pchar* refers to a "primitive character", from the ASCII character set, and requiring one char location of storage. In the UTF-8 encoding, other, "extended characters", occupy between 2 and 4 character locations.[2]

The current Standard description of `ACCEPT` has the signature:

```
( c-addr +n1 -- +n2 )
```

and a description that begins with: "Receive a string of at most +n1 characters..". This needs to be re-expressed as "Receive a string of length at most +n1 address units". There are further issues if markup is used to enter extended characters. After entering "`\alpha \beta` " we have the two char-

[2] ,This is not quite accurate, since, in some alphabets, characters may be written with diacritical marks which themselves are encoded as separate characters. However, we already have enough to deal with here..

acter string $\alpha\beta$, which of four address units in length. However we have to allow a buffer length of ten in order to enter the string. Where a *pchar* occupies one byte and UTF-8 encoding is used, the maximum character length is 4 bytes. If m is the length of the longest markup sequence and $m > 4$, then to allow entry of n characters it is sufficient to accept a string of length $4 * (n - 1) + m$ address units.

`ACCEPT` is typical of a number of Standard words which deal with characters and whose glossary definitions require rewording in terms of character locations rather than characters. A new wordset is required to deal with the special operations required of wide characters, and here the current RFD provides valuable suggestions.

7 Conclusions and future work

Using markup to access a limited range of special characters allows us to write Forth code which can makes use of classical mathematical symbols. This is particularly beneficial to RVM-Forth which has an extensive set package and supports λ notation. Future work includes rewriting the mathematical parts of the RVM Forth source code using the extended character set where appropriate, and writing infix expression compilers for set expressions using the techniques outlined in our EuroForth 2008 paper "Using Forth in a Concept Oriented Computer Language Course." Further work is also envisaged to get the Forth IDE words `SEE` and `EDIT` working with more editors, and combining this functionality with that of the markup filter.

Porting MINOS to VFX

why source inlining sucks

Bernd Paysan

EuroForth 2008

Outline

1. Motivation
2. Porting MINOS
   - Porting OOF
   - Porting MINOS
3. Status

Motivation

- Stephen kindly asked me to port MINOS to VFX
- Making MINOS more portable is in my intention, too
- VFX is sufficiently different from bigFORTH and I can't easily add everything I need ⟶ gives more benefit than porting to Gforth first
- Potential customers with dual–license (closed source customers have to pay)
- Lessons to learn about more optimizing Forth systems

What has to be done?

- Port OOF to VFX — there's an ANS version of OOF, but it is quite slow, because it uses a lot of CREATE DOES>
- Change the X lib calls to conventional parameter order
- Supporting libraries missing in VFX Forth

Porting OOF

- Create method invocation and instance variables with code that looks like this:

```
: (method, ( offset -- )  >r
  : r> o@+,
    postpone @ postpone execute postpone ; ;
```

- Problem: Source inliner sees no source, and compiles nothing
- Solution: Add `discard-sinline` before `postpone ;`

Porting MINOS

- Convert MINOS source from block to files
- Lot of MINOS code is backend–specific, i.e. lots of `[IFDEF] x11` and `[IFDEF] win32`
- Replace with `[defined] x11 [IF]` (part of Forth200x), because VFX has no `[IFDEF]`
- Change base–specified numbers to Forth200x proposal

Alias vs. Synonym

- Problem: OOF declares early binding methods before definition
- Solution in bigFORTH: Special header bit for aliases that can be changed later
- Solution with Synonym (ugly dirty hack) is not possible
- Better attempt: Call dummy word, and replace call offset later
- Side effect: One more item on the return stack
- Future solution: Jump to dummy word instead

Iterators

- MINOS uses some iterators in the following form:

```
: ALLCHILDS ( .. -- ..' )
  childs self
  BEGIN dup 'nil <> WHILE
       r@ swap >o execute widgets self o>
  REPEAT drop rdrop ;
```

- Preventing ALLCHILDS to be inlined is easy
- Problem: ALLCHILDS poisons source inliner:

```
: >hglue  ( -- min glue ) 0 0  ALLCHILDS hglue@ p+ ;
```

Inlining >hglue will not work!!!

- Solution: Disable source inliner for most of the time (iterators are used too often)

Event Loop

- In bigFORTH: Cooperative background task — works well, no locking necessary
- VFX Forth has a preemptive multitasker (native thread): This doesn't work
- Adding the event loop to keyboard event check didn't work so far
- Interim solution: Have a special `event-loop` word

Status

- The calculator example works
- Some complex classes (e.g. OpenGL) not yet ported
- Theseus needs porting, too
- More talk with Stephen needed to resolve some problems

A high level interface to SQLite

Federico de Ceballos
Universidad de Cantabria

Some sample SQL code

```
begin;

create table episodes (id integer primary key,
                       season int,
                       name text);
insert into episodes values(1, 1, 'male unbonding');
insert into episodes values(2, 1, 'the stake out');

create table foods (id integer primary key,
                    type_id integer,
                    name text);
insert into foods values(1, 1, 'bagels');
insert into foods values(2, 2, 'bavarian cream pie');

drop table foods;

commit;
```

The SQLite interpreter

```
SQLite version 3.6.1
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> begin;
sqlite> create table episodes (id integer primary key,
   ...>                        season int,
   ...>                        name text );
sqlite> insert into episodes values(1, 1, 'male unbonding');
sqlite> insert into episodes values(2, 1, 'the stake out');
sqlite> create table foods (id integer primary key,
   ...>                     type_id integer,
   ...>                     name text );
sqlite> insert into foods values(1, 1, 'bagels');
sqlite> insert into foods values(2, 2, 'bavarian cream
pie');
sqlite> drop table foods;
sqlite> commit;
```

Types of words

- Some words are not used in Forth: **commit insert**
- Others are: **begin create drop**
- A few words may appear by themselves, without additional parameters, so the closing semicolon could be attached: **begin; commit;**

Getting results

After a normal query, we expect to receive a result set. This is usually printed on the screen.

Some words allow the user to choose the format used.

```
+headers      A header is produced
-headers      A header is not produced

mode-csv      Columns are separated by a string
mode-column   Columns are of a given width
mode-line     Each column is given in its own line

set-separator Sets the string used as separator
set-null      Sets the string used for null values
set-widths    Sets the widths to be used for columns
```

Using parameters

```
insert into foo values(?,?,?)";p

   [ 1                    1 int]
   [ s" pi"               2 text]
   [ 1e fatan 4e f* f.    3 float] ;p

   [ 2                    1 int]
   [ s" e"                2 text]
   [ 1e fexp              3 float] ;
```

The [word is used to “pop” out of SQL mode and into Forth mode, in a similar way as you are able to temporally leave compilation state to go to interpretation state.

Other possibilities

```
mode-user     A user function is called for each row, this
              function has to get the column values

mode-stack    A user function is called for each row with
              the column values already on the stack

: sample ( ) cr 1 get-text type ;

/sql

' sample mode-user

select * from my_table;

sql/
```

Defining user functions

```
: sample ( ) 0 get-int 1 get-int + result-int ;
: sum    ( ) 0  #args 0 ?do  i get-int +  loop  result-int ;

/sql

' sample  2 def-function my_function
' sum    -1 def-function sum

select my_function(1,2);
select sum(),sum(1),sum(1,2);

sql/
```

Using code inside definitions

```
s" insert into episodes (id) values (?)" prepare
   30 1 bind-int
continue
   35 1 bind-int
conclude

:noname ( ) cr 1 get-int . ; is row

s" select * from episodes" process

s" insert into episodes (id) values (" >sq (.) +sq s" )" +sq
sq@ process
```

This is normal Forth code that can be used anywhere.

Future work

Test the code, complete the binding and make it public

Expand the system by a new set of functions

Move files into a database

Use the program as part of a course ?