Popit: Implementing a Forth-like Language in C and Scheme as Student Projects

Bill Stoddart, Robert Lynas, Frank Zeyda
School of Computing and Mathematics
University of Teesside, Middlesbrough, U.K.

November 10, 2004

Abstract

As programming environments and operating systems grow in complexity, students of computer science find it increasingly difficult to gain a holistic understanding of the computer systems they study. "Popit" is a Forth-like language designed to be implemented by students in order to expose them to a complete, though simple, programming system and virtual machine. The Popit implementation exercise is used in a group project module in the second year of the course, where it is implemented in C. In the fourth year of the course it is offered as an individual project to be implemented in Scheme.

keywords: Programming, Student Projects, Forth, C, Scheme

1 Introduction

Over a generation the complexity of the computing environment faced by students has changed by several orders of magnitude. From small machines with simple software support and fully described hardware interfaces, we have evolved to the use of complex operating systems where even simple functionality is wrapped in millions of lines of code. The effect of these changes is that students have a less holistic understanding of the computing environment and "subject knowledge" appears fragmented and arbitrary, rather than based on fundamental principles.

In an attempt to counter these trends we have developed project work in which students implement a simple stack based virtual machine and programming environment which we call "Popit". Popit draws heavily from the directness and simplicity of Forth. Second year students implement and program in Popit as part of a group project assignment in C. Final year students may choose the implementation of Popit in Scheme as an individual project.

The paper is organized as follows. In section 2 we describe aspects of the Popit language. In section 3 we give some examples of Popit programs. In section

4 we describe the C version of the project. In section 5 we discuss project management and assessment. In section 6 we describe the Scheme version of the project and in section 7 we draw our conclusions.

2 The Popit Language

Popit is a close relative of Forth and uses an identical stack organization with separate parameter and return stacks. Where possible it uses the same operation names as Forth. However, it is less ambitious in terms of extensibility and less crash prone.

Consider this standard Forth code which declares a variable `X`, sets its value to 100 and then prints its value.

```
VARIABLE X   100 X !   X @ .
```

Here use of `X` pushes the address of `X` onto the stack. Accessing the memory at this address requires use of `@` "fetch" or `!` "store". In Popit these words are not available. Our approach is that variables must be responsible for storing or returning their associated values. For simple variables, we can remain within standard Forth whilst doing this by use of `VALUE`. An example usage which achieves a similar effect to the above is:

```
VALUE X   100 TO X   X .
```

By adopting this style we avoid invalid memory access arising from careless use of raw memory access words, but we lose some powerful primitives which can used in Forth when creating new defining words. For example in Forth `VALUE` itself can be defined with these primitives.

In Forth, control structures such as `IF  ELSE  THEN` can be provided by defining the key words of the structure *in Forth itself* as immediate words: words which act immediately during compilation rather than being incorporated into a new definition. This is another aspect of extensibility that we do not include within Popit, where `IF  ELSE  THEN` are built-in key words. Each token in Popit is first checked to see if it is a key word, of not it is then checked against the names of operations, and if a match is still not found an attempt is made to interpret it as a number. An effect of this scheme is that key words cannot be redefined. The possibly unwise definition `: IF 2 ;` will be accepted by Popit, but never seen, as `IF` is always seen first as a key word.

3 Popit Programming

Despite the limitations mentioned above, we want students to appreciate the inherent extensibility of postfix notation, and to this end we provide a limited number of condition-tests which they are invited to extend with definitions such as:

```
: >= ( n1 n2 -- f,  f is true iff n1 greater than or equal to n2 )
```

```
  < NOT ;
```

We emphasise, by comparison with C, how perfectly such new operations integrate into the language.

Local variables are not provided in Popit, but we can do quite well without them by defining a user stack.

```
10000 CONSTANT USER-STACK-SIZE
USER-STACK-SIZE VALUE-ARRAY USTACK
VALUE USP   1 TO USP

: PUSH ( n --  push n onto user stack, PRE: USP < USER-STACK-SIZE )
  TO [ USP ] OF USTACK ( move n into the user stack )
  USP 1 + TO USP ( increment the user stack pointer ) ;

: POP ( -- n,  pop n from the user stack, PRE: USP > 0 )
  USP 1 - TO USP ( decrement the stack pointer )
  [ USP ] OF USTACK ( move n to the data stack ) ;

: USEMPTY ( -- flag ) USP 1 = ;

: USFULL ( -- flag ) USP USER-STACK-SIZE = ;
```

The user stack allows global variables to be borrowed for a local purpose. We illustrate this in the following "Towers of Hanoi" example. We have three poles, 1, 2 and 3. We have to move a pile of discs from one to another via the third, moving one disc at a time and never placing a larger disc upon a smaller one. Note that, unlike Forth, recursion is expressed by invoking an operation by name within its own definition. The definition of `HANOI` uses the variables `N`, `FROM-POLE` and `TO-POLE` at each level of recursion, using the user stack to save and restore their previous values.

```
VALUE N   VALUE FROM-POLE   VALUE TO-POLE

: VIA ( -- via, where FROM+TO+via=6 ) 6 FROM-POLE TO-POLE + - ;

: MOVE-DISC ( from to -- )
  CR
  ." Move disc from pole " SWAP .
  ." to pole " . ;

: HANOI ( ndiscs from to -- )
   N PUSH  FROM-POLE PUSH  TO-POLE PUSH
   TO TO-POLE  TO FROM-POLE  TO N
   N 1 =
   IF
     FROM-POLE TO-POLE MOVE-DISC
   ELSE
     N 1 -  FROM-POLE  VIA  HANOI
```

```
  FROM-POLE TO-POLE MOVE-DISC
  N 1 -  VIA  TO-POLE  HANOI
THEN
POP TO TO-POLE  POP TO FROM-POLE  POP TO N ;
```

4 The C Project

The project is done in teams of four students. The project is broken down into a number of components: text input (keyboard or file), lexical analysis (same as Forth's), numeric i/o conversions, virtual machine components such as memory and inner interpreter, virtual machine operations, and the operations table, handling the lookup and addition of new operations.

We have run the project in different ways, either letting students take fairly complete responsibility for design, or, at the other extreme, providing object code for each program component and requiring them to write code that exactly duplicates its function (which is also carefully specified). Neither approach is best in all ways or for all students. Our instincts would be to allow maximum freedom, but the lack of grounding we observe in our current students means we have moved completely to the second approach.

Because we are interested in students getting to grips with fundamentals of programming, we want them to write from the ground up, rather than using the C library. However, banning the use of the library could send a confusing message in the context of their overall development, so we don't go that far. Instead we specify system components in such a way that the library routines do not quite do what is required. For example the C library routine `strtol` will perform string to integer number conversion in any base from 2 to 36, but it accepts some special forms such as 0xnnnn for hex numbers, and these special forms are not part of the Popit language.

In addition to object code, students receive a mixture of complete and incomplete C source files for the more tricky parts of the project. This includes the virtual machine memory components and threaded code inner interpreter. The virtual machine has both program and data memory, and a `primitives` table containing the C functions which implement primitive operations such as SWAP, DUP, + ...

The execution token of each Popit operation is an integer giving the location within the program memory array where an entry for that word is found. For primitive definitions (those coded directly in C) this location contains the position of the primitive in the primitives table. For high level definitions it contains the position of the `nest` primitive, whose role is to save the current virtual machine instruction pointer and set this pointer to the virtual machine code that follows. This code contains the tokens for the compiled code of the definition. Compilation of the definition `: >= < NOT ;` lays down four tokens, for `NEST`, `>=`, `NOT` and `EXIT`. The corresponding C functions can be coded as:

```
void nest() { rpush(ip); ip=w+1; }
```

```
void greaterthan() { int n2 = pop(); int n1 = pop();
   if (n1 > n2) push(TRUE); else push(FALSE);
}
void bitwisenot() { push(~pop());}
void eggzit() { ip=rpop(); }
```

An indirect threaded code inner interpreter is used to wind execution through compiled code. In the following `pmem` is the program memory array, `pp` is an integer which gives the next free location in that array, and `ip` is the virtual machine instruction pointer.

```
void execute(int opindex) { /* execute the command with the given opindex */
   /* set up a temporary pointer in the first free element of pmem */
   pmem[pp]=opindex;
   ip=pp;
   /* Enter the inner interpreter execution loop. When we exit the command ip
      will be incremented by 1, i.e. equal to pp+1. Also ip is guaranteed never
      to take this value before execution of the command terminates, as it can
      only range over the already allocated region of pmem */

   while(ip != pp+1) {
      w = pmem[ip];
      ip=ip+1;
      primitives[pmem[w]]();
   }
}
```

As will be seen, we do not use the most concise style of C coding. We rather wish to show what happens at each step. Since students are given this code, we require them to produce some example traces to demonstrate their understanding of its operation.

For the compiler, students are given part of the source code. For example we provide the source code to handle `IF` constructs, but the students have to give the code for loops. The code for `IF ELSE THEN` is made up of compiling operations and syntax checking:

```
/* define syntax tokens for key words */
#define IF     100
#define ELSE   101
#define BEGIN  103

/* compiling operations performed in response to IF ELSE and THEN */

void IFcomp() {
   compile(zbranch_i); push(pp); compile(0);
}

void ELSEcomp() {
```

```
   compile(branch_i);
   pmem[pop()]=pp+1; /* resolve branch address of previous IF */
   push(pp); /* leave the branch address of ELSE for resolution by THEN */
   compile(0); /* compile a filler into the branch address */
}

void THENcomp() {
   pmem[pop()]=pp;
}

/* Syntax checks for .. IF .. [ELSE] .. THEN .. */

void IFsyntax() { spush(IF); }

void ELSEsyntax() {
   char errmess[100] = "ELSE with no matching IF in defn of ";
   if (spop() != IF)
      reporterror( strcat(errmess,lastname()) );
   else {
      spush(ELSE);
   }
}

void THENsyntax() {
   int syntax;
   char errmess[100] = "THEN with no matching IF or ELSE in defn of ";
   syntax=spop();
   if( !((syntax==IF) || (syntax==ELSE)) )
      reporterror( strcat(errmess,lastname()) );
}

/* combining compiler actions and syntax checks for IF, ELSE and THEN */

void doIF() { IFsyntax(); IFcomp(); }
void doELSE() { ELSEsyntax(); ELSEcomp(); }
void doTHEN() { THENsyntax(); THENcomp(); }
```

All immediate and defining words are associated with the functions that perform their associated actions through a lookup table:

```
typedef char name[32];
typedef void (*action)();
typedef struct keyword_entry { name namefield; action key_action; };

/* now declare and fill the table */
keyword_entry keytable[] = {
   { "VALUE-ARRAY", doVALUE_ARRAY },
   { "VALUE", doVALUE },
   { "CONSTANT", doCONSTANT },
   { "IF", doIF },
```

```
{ "ELSE", doELSE },
{ "THEN", doTHEN },
...
```

This is the table of Popit key words (mentioned earlier). These take the place of immediate and defining words in Forth, with the consequence that these are beyond the scope of Popit definitions, being provided once and for all by the Popit compiler. This is not because we prefer this approach to that of Forth, but because we are designing a project in C programming to run over a limited time duration in which the finer points of Forth programming will not be part of the agenda.

5 Project Management and Assessment

At the start of the project students receive a description of Popit which includes a specification of each operation. They receive a specification for each C component in the Popit implementation which again contains a specification of each function. They receive all object code files for the Popit executable and some parts of the corresponding source code.

They are first required to link and run the Popit executable, and to write some simple Popit programs. This gets them familiar with the system they are going to have to produce.

Next we require them to take the object code files along with the specifications of the C functions they implement, and to write test harnesses for these files. One reason we find this step necessary is that many students have very little idea of what has to happen in any system. For example, they find it difficult to appreciate the need for conversion between character string and internal binary representation of numbers. These students are not stupid, but they have had to take on a wider agenda than students in the past. They tend to spend a lot of time on installation and configuration issues, with the result that they are less *au fait* with the basics of programming.

After the testing phase each students within a group takes on responsibility for coding parts of the project. Assessment is by a combination of report and project viva, at which code is demonstrated and each student is asked a mixture of detailed questions about the code he is responsible for and more conceptual questions about the project as a whole.

The project has been taken by between 60 and 80 students a year for the past four years. A possible problem with such an arrangement is collusion between current students and those who have previously done the work. This has not been a problem so far, perhaps because following this project students are placed in industry for a year. Such collusion would be noticeable, because each year we make changes to the project in the light of previous experience. For example having noted that students find the keyboard input component relatively easy we added the requirement for a command line history buffer.

6 The Popit Project in Scheme

A noticeable aspect of the C version of Popit is its very literal interpretation of the phrase "Virtual Machine". The environment constructed was almost a simulation of a physical computer design, à la von Neumann, but reproduced in software. We have large integer arrays used for program and data memory, echoing the linear ordering of actual memory with sequential location addresses, a program counter, and flow control structures implemented with jumps to other program memory locations.

The contents of these "memory" locations can be literal numbers (or character codes), pointers to other locations, or addresses in the optable; these would be used to invoke primitives or earlier definitions. Program execution was achieved by stepping through the instructions and/or data contained in the locations currently pointed to by the program counter.

This is a good vehicle for inculcating a kind of hands-on understanding of the way a real computer operates, if one views the C-coded primitives as processor instructions. The functional, lambda-inspired model, on the other hand, provides a more abstract way of thinking about Popit definitions, variable values and stack. Our experience with this project is limited to one student, whose initial response was a design which reflected the C model. However, with a little coaxing our student was soon able to provide a far more abstract approach.

An essential feature of the lambda calculus, the mathematical formalism which underpins scheme, is that functions are regarded as "first class objects", that is, they may be bound (rather than "named" in the C sense, as one cannot have an anonymous function in C) to a name, re-bound to another name, exist anonymously, and passed as arguments to another function as actual applicable objects. Also, a directly applicable function object can be returned from another function. This implies a level playing field between functions and other program objects; literals or variables. In accordance with this outlook, Scheme (unlike Common Lisp, for instance) uses the same kind of namespace for function names as variable names, so one can simply refer to a function by name, and apply it by putting brackets around it (along with any arguments required). There might be a superficial syntactic resemblance between this and the way a C function name denotes its address and the way it is applied by putting brackets/ args in front, but the underlying semantic concept is quite different.

One consequence of this is that sequential (and of course, functional) composition becomes very easy and elegant. The heart of Scheme Popit compiler is this tiny function:

```
(define (compose f g) (lambda () (f) (g) ))
```

which takes two functions, binds them to the names f and g, and creates a "closure" representing the sequential composition of the functions.

The approach to definition building is incremental. Each new operation (primitive or existing definition) is sequentially composed onto an existing function, making a single new function. The process starts with an empty function (no-op). A completed Popit definition might be a primitive, referring to a Scheme

function, or a composition of several of these and possibly other previous definitions, but still only a single function.

A function can also be a nameless block of actions occurring arbitrarily in the middle of a function definition process – that is, while a new definition is being compiled. For instance, the body of a while-loop, and the if-body and else-body of a conditional can be viewed as (and indeed are) separate functions, albeit ones which make little sense outside their context.

This notion is used to implement conditionals and loops with a definition stack, the current definition block being always at the top of the stack. A no-op waits in readiness if no compilation is happening. During a compilation, if the compiler encounters an IF, a new no-op will be pushed onto the def stack, in preparation of a new block of incoming operations, which will form the body to be executed if the IF-test (of the top of the Popit stack) is true. If the compiler subsequently encounters an ELSE, another no-op is pushed to form the foundation for the else-body, to be executed if the IF-test is false. Since this is optional, a no-op is used for the else-body where none was specified.

On receiving the THEN token, the compiler uses the contents of the stack to build an IF construct (a single function), and then compose this onto the existing definition, which is still at the bottom of the stack. In pseudocode, the construction of an if-block looks like this:

construct-if (true-body, false-body) lambda () if Popit-stack-true apply true-body else apply false-body endif end-lambda end

The lambda keyword in Scheme introduces an anonymous function definition, which here forms a closure with the true-body and false-body supplied to the construct-if function. This lambda function is what is returned by the construct-if function. Thus the respective bodies will not be actually applied until the function is called (during the execution of its parent definition), and the Popit stack tested. The returned function here is then compiled into the parent definition, leaving the complete def so far at the top of the def stack.

The overall approach adopted is not absolutely "pure" in the functional programming sense. A pure approach would have no assignment statements, and therefore no concept of a "state" which persists outside of (if not independent of) the function applications which move the computation along. Being a Forth-like language, Popit is inherently blessed with state in the form of its stacks, along with a small number of variables, e.g. that representing the current number base. The most puritanically functional implementation would therefore have to regard these as a collective "environment" which would have to be passed on – at least the relevant parts of it – between each function application as a parameter.

However, Scheme does provide a primitive of its own for destructive reassignment (called set!), which mostly allows the updating to be hidden away inside functions. For example, the stacks can be implemented as functions which store their own internal state, i.e. as abstract data types. Even an integer variable, using this style of programming, is actually a function storing an integer as its internal state, which can be accessed or changed by the appropriate function

call. Thus there is no notion of variables being locations in memory – we have abstracted away the idea of value storage from that of storing values in a particular place, and instead entrusted them to an anonymous function bound to another function whose job it is to relay interrogations or updates.

On the whole, the approach makes, at least conceptually, a simple and elegant alternative way of thinking about Popit implementation, certainly for those students previously only exposed to the strictly imperative, sequential languages. To be sure, these features are far from entirely absent from Scheme and are used in our Popit implementation. We were, however, able to abstract many virtual machine components, including program and data memory.

The abstraction of data was made easier by the absence from Popit of Forth memory access primitives such as `@` and `!`. The use of `VALUE` declarations, on the other hand, lends itself readily to expression in Scheme.

7 Conclusions

In a world of increasing complexity the projects described here give students a rare opportunity to implement a whole system consisting of a virtual machine and an associated programming language. The C and Scheme flavours of the project use very different levels of abstraction in accomplishing this task.

In the group project, students are introduced to the science of working effectively as a team. Since each component must link and execute as part of the overall system, students have to play close attention to the component specifications. These define a boundary between creativity and discipline in that there is absolute freedom allowed as to *how* each specification is met, and absolutely no freedom to deviate from the specification. This is re-inforced by providing the students with a working solution in the form of object code files which can be linked to produce the Popit executable. They must then replicate these components from the given specifications in such a way that any object code file they produce may be substituted for the corresponding provided object file without compromising the linking and execution of the Popit system.