Purrr

TOM SCHOUTEN

November 2, 2007

Abstract

Purrr is a Forth dialect specifically tailored to flash ROM based microcontrollers. A Forth typically enables direct low–level machine access in a resource–friendly way while providing a solid base for constructing high–level abstractions. Purrr includes a purely functional compositional macro language for meta–programming. The Purrr implementation consists of an optimizing cross–compiler and a live interaction interpreter, both running on a host PC system. Purrr's live interaction interpreter supports an incremental, bottom up programming, testing and debugging style. The current Purrr implementation supports the 8–bit Microchip PIC18 architecture.

1 Introduction

This manual documents the Purrr *macro* language and its *live interaction* system. Purrr is a Forth dialect built as a collection of composable macros implementing a *stack machine model* as a thin layer atop a concrete machine architecture. The interaction system provides a command console for live machine interaction and incremental compilation of Forth code.

This manual is intended for an audience somewhat familiar with Forth and assembly programming. It is not necessary to be an expert in either. If you are looking for a tutorial, have a look at the Purrr[1] website.

1.1 Expansion and Contraction

To understand Purrr from a high level, it is good to start with the *substitution model* of code translateion. A Purrr program is essentially a collection of macros which will be *instantiated*: it will eventually be translated to *executable machine code.*

A string of *words*, which are textual entities separated by whitespace, can be *expanded* to another string of words by *substituting* each word with its definition. This process continues indefinitely until a word can no longer be expanded into a string of words. Such a word is called a *primitive.* These primitives have a direct relation to executable machine code. For example, the word `double` might expand to the string `dup +`.

Thus far this model describes just the semantics of a *macro assembler.* Purrr is an extension of this paradigm because it allows a process of *contraction* too. For example the string `1 +` might contract to the word `increment`.

So substitution words both ways: words can be expanded to strings, but strings can also be contracted to words, or different strings. This model is simple but powerful. It allows a unified approach to the problems of *optimization* and *metaprogramming.* The process of expansion supports *bottom up programming*: building larger things out of smaller things. The process of contraction allows *program specialization*: using general high level components to implement specialized efficient code.

These processes are implemented using *macros.* A macro is merely a function that, at run time, will perform the steps of expansion and contraction, resulting in an *incremental* compilation process that can be read from left to right[1].

1.2 Interaction

Traditionally, Forth is implemented as an interactive self–hosting system. Here self–hosting means it can extend itself by translating source code to some executable representation. Purrr however is *cross compiled.* The compilation runs on a separate *host* computer. In order to keep the interesting interactive properties of a self–hosted Forth system, some effort has been spent to provide at least the *illusion* of a self–hosting system: code compilation and upload is possible while a system is running.

[1]This might not be an optimal implementation since it is greedy, but is a lot easier to implement than a search based solution.

1.3 Rationale

At the moment of writing, Purrr is implemented for the Microchip PIC18 architecture. Purrr is not a standard (ANS) Forth. It is fairly minimal, and takes from Forth just those elements that are essential to build such a proto language: procedure and macro words, stacks, global variables and a couple of control structures. The key differences are the use of 8–bit data words, separate code and data spaces, and the lack of a self–hosting interpreter and compiler.

Purrr supports a reasonable high level of programming while still retaining precise control over the underlying machinery. This is necessary in the intended problem domain: microcontroller programs often contain a mix of highly specialized low–level code that has to be as efficient as possible, and a bulk of high–level management code that is complicated but often less time critical. Forth stacks enable a referentially transparent programming style resembling *functional programming*, where a program is factored into a large collection of small functions called *words*. Such a style promotes reuse, layered abstraction and individual testability. In the spirit of Forth, the Purrr Forth dialect is meant to be extended to a language matching a problem domain. This approach is called *bottom up programming*.

Purrr aims to be minimal from an on–target kernel perspective: it is a native Forth, not requiring a runtime kernel. Effective boot code is only a couple of bytes to setup the stacks.

Purrr extensively uses macros as a building block. A macro an abbreviation that is inlined at the place of occurance. However, macros in Purrr can *recombine*. Purrr macros are composable, just like functions, and can be used to add language features that cannot be expressed using composition of procedure words alone. Purrr's macros make up a purely functional compositional stack language language, for which the substitution principle holds, and exhibit a rich type system that can be used to perform compile time computations.

Considerable effort has been spent on keeping the system interactive, and reasonably *introspective*, just like a self–hosting Forth. It is possible to inspect and modify machine code and data state while running, execute arbitrary code, and compile and upload code on–the–fly. This is an essential element of a Forth development system and should not be lost due to target size limitations or the absence of an on–target interpreter.

2 The Purrr Language

2.1 How Forth?

A Purrr program consists of a sequence of *words*. There are two classes of words. A *procedure word* refers to a program fragment that is represented as an individually executable chunk of machine code instructions. A *macro word* is a function that represents a compile time action, which eventually results in machine code. In this manual we abbreviate these names to *word* and *macro* respectively.

```
macro
: increment  \ n -- n+1
    1 + ;

forth
: double-increment \ n -- n+2
    increment
    increment ;
```

The words `macro` and `forth` switch between macro word and procedure word definitions respectively. In the code above, `increment` is a macro while `double-increment` is a procedure word. The backslash character `\` is used to start line comment.

Following Forth tradition, Purrr's procedure words have a low–level concrete semantics. Purrr is essentially a macro assembler. There is a fairly direct relationship between a Purrr program text and compiled machine code.

The low–level semantics is complemented with a powerful macro language. The programmer can influence the relationship between source code and native machine code by writing new meta–programming constructs, in the form of purely functional *composable* macros. Compared to a traditional macro assembler, macros do not only expand to asembly code, they also *recombine* with previously generated assembly code.

In the example above, the procedure word `double-increment` corresponds to the code

```
double-increment:
       addlw  (1 1 +)
       return 0
```

Purrr is special compared to standard, explicitly meta–programmed Forth, because its metaprogramming through macro composition can be understood as *partial evaluation*, an optimization technique that performs program specialization. In Purrr this idea is stretched to the point that some programs *need* to be specialized in order to be executable (compilable) at all.

Compared to standard Forth, Purrr thus uses a simplified *implicit* metaprogramming syntax. Standard Forth uses *explicit* metaprogramming in the form of the words `[` and `]` which switch between compile and interpret mode. In Purrr, the programmer does not explicitly indicate which code will run at compile time.

The example above illustrates the interplay between partial evaluation and the concrete semantics of Purrr (its relationship to machine code). The double occurence of the word `increment` has been partially evaluated to the machine operation `addlw (1 1 +)`. The code between parenthesis indicates a function that can be evaluated at compile time, here producing the numeric value 2. The machine instruction `addlw` ADDs its Literal argument to the Working register representing the top of the data stack.

Partial evaluation is an optimization technique often used in the implementation of functional programming languages. This approach works for Purrr because it is possible to interprete a *subset* of the procedural Forth dialect as a purely functional compositional language. The time at which function evaluation by composition occurs then becomes a parameter to play with: this makes it possible to move some of the evaluation to compile time, as is shown in the example above.

The metaprogramming power lies in this compile time evaluation. Let's illustrate this for arithmetic by going back to the example `(1 1 +)` above. The integer operation `+` when it is done at compile time has infinite precision. The same goes for the other integer arithmetic operations. In order to be able to represent the result on the target, results of computations need to be truncated to the data word size, which in case of Purrr18 is 8 bits for data and 16 bits for code addresses. This technique enables the use of arithmetic operations that are not available at run–time in a way that is fairly transparent: it is possible to *read* source code looking only at the high level meaning of code, without worrying whether constructs are *compilable*.

In order to effectively *write* programs, the programmer does have to worry about whether a certain construct is compilable. In practice however, this is quite straightforward. One way of looking at the approach is to view procedural Purrr as the *projection* of a clean purely functional, compositional, high–level language, onto a restricted procedural semantics. See the Brood paper[3] for a formal treatment of this relation.

Summarized, the important property of macros is that they can be composed, and such compositions can be partially evaluated to yield compilable constructs. The partial evaluation of arithmetic expression is but one example of this powerful construct. By relaxing the requirement that all macros need to be *compilable* in isolation, one can use macros to construct language *idioms*. Idioms are sequences (compositions) of words that yield some compilable construct. A non–compilable construct is called *ephemeral*. An example of an ephemeral macro is `begin`. It is not compilable without a balanced `again` or `until`. This approach enables the use of very high level compile–time operations as long as they eventually lead to constructs representable in low–level form, or can be *reduced* to some representable construct, as is the case for numbers.

In some sense, this projection turns the Purrr language into a *leaky abstraction*: the programmer has to be aware of what functionality is lost in this projection. Purrr can be seen as an instance of the *human compiler* anti–pattern: the programmer deals with the reduction of the high level Purrr macro semantics to low level compilable constructs, by deciding which operations are to be implemented as procedure words. In exchange for this manual labour, one gets direct access to hardware in an environment that enables a lot of highlevel programming constructs in slightly restricted form. This idea is almost entirely stolen from PreScheme[2]. Procedural Purrr is to macro Purrr what PreScheme is to Scheme.

2.2 Tool Chain

In the Purrr tool chain, the meta–programming and code generation occurs on a system which is *different* from the one executing the final machine code. Two computer systems are involved: the *host* system runs a compiler program to produce compiled programs from source code while the *target* system eventually executes these compiled programs. The main reason for this distinction is of course the lack target resources to support the tool chain.

The host–target distinction is important from the point of *interaction*. Procedure words exist physically on the target chip in the form of machine code, and can be *executed* interactively. Macro words exist only in the translation phase from source code to machine code, and have no direct representative as an accessible code word, and as such cannot be executed. However, Purrr includes the possibility of executing macros that produce constant values, as if they were present in compiled form. Similarly, some basic arithmetic operations are simulated if they are not instantiated as machine code.

2.3 Why Forth?

The most compelling property of Forth is its ease of performing *composition*: syntactically, a program is merely a concatenation of the names of sub–programs, represented as words. If a sequence of words occurs in more than one place in a program, one can give a name to the sequence, and substitute the occurrence of the sequence in the source code by the newly defined name. This technique of program evolution is called *(re)factoring*, and is essential for controlled growth.

In short, when a pattern emerges in the source, it is time to increase the abstraction level and provide some *correctness preserving program transformations* to isolate the code patterns and give them a name. In Forth this usually means to change the order of some words so a sequence can be cut out and replaced by a single name referencing a procedure or macro[2].

Factored procedure words are important because they allow physical (on chip) code reuse, which limits the necessary target code space. Forth is famous for doing a lot with a little, exactly because of its high affinity for factored code. Forth is a *compression algorithm*. It is probably not a coincidence that conciseness and elegance are correlated.

Factored macro words are important because they allow the construction of language features that are not expressible as a composition of procedure words. In Purrr, macro words can be composed just as easily as procedure words. A category of words necessarily implemented as macros are *control words* which change the flow of control to something else than the default sequential word execution. Another example is *optimization*; some macros can be combined to code that is simpler or has more efficient representation than the sum of the parts. A third example in Purrr is the use of *idioms*, which are sequences of macro words that behave as if they were simply composed words, but have only a meaning when combined in a certain way, allowing the expression of constructs that are impossible to express as procedures.

[2]Being aware of patterns is what programming is all about. It is important to see patterns in your problem, so you can divise a feasible solution. However, it might be *more* important to close the loop and see the patterns in your *solution*, so you can bring your understanding of the problem to a higher level.

In Purrr, compile time computations have access to a type system that is substantially richer than the raw machine words used at run time. This type system is Scheme's. Notable types are infinitely precise integers and rational numbers.

3 The Purrr Programming Model

Purrr is a compiled language, and works without a run–time kernel. A Purrr program is defined in terms of *composable macros*. Compilation of a Purrr program is a function which maps a *source file* and a *dictionary* to an updated dictionary and a chunk of binary machine code. It is factored into the following steps:

- Parsing of program text into macros and procedural code.
- Construction of an extended compiler from the base compiler and the named macros.
- Compilation of the code body to assembly language, using the extended compiler.
- Construction of dictionary items for the procedural code, and assembly of binary machine code, statically bound to functionality represented by the updated dictionary.

The *dictionary* is used as a representation of an *abstract* collection of procedures. These procedures operate on the machine state, and contain a functional subset operating on a parameter stack. They are implemented as executable native machine code. Purrr's programming model is *strictly bottom–up* and *early bound*. Purrr programs are strictly layered, with layers being compiled separately. Upper layers cannot influence functionality in lower layers, unless this is explicitly permitted by some late binding mechanism (implemented as an add–on).

Purrr uses *partial evaluation* as an interface to the metaprogramming system. This is implemented using greedy macros.

4 Two Kinds of Macros

Essentially, there are two kinds of primitive macros. Those that operate on the *compilation stack* and those that operate on the *macro stack*.

4.1 Partial Evaluation

In order to see how partial evaluation works, it is a good idea to look at how it is implemented. In the transcript below I show the effect of incremental compilation. Compilation works by pushing data on a *compilation stack*. The data on this stack is *typed*, with the type indicated by a symbolic prefix.

We start with entering a number

```
>> 1
        qw      1
```

The first line is the compilation input, the remaining lines are the contents of the compilation stack, which is printed using the command `pa`. The type `qw` indicates a Quoted Word. In order to be compilable, the word needs to be reducable to a numeric value. We go on by entering another number.

```
>> 2
        qw      1
        qw      2
```

There are now two numbers on the compilation stack. Next we enter an operation.

```
>> +
        qw      (1 2 +)
```

The result is a quoted word, where the word can be reduced to a number by evaluating a computation. This is the simplest example of a compile–time computation.

When a compilation is done, all the data left on the compilation stack needs to represent a compilable program. In this case, we have a single quoted number 3, which is certainly compilable. Let's start over with a clean compilation stack and type just the operation.

```
>> +
        addwf   POSTDEC0, 0, 0
```

This is quite different. What is present on the compilation stack is an assembly program that will perform the computation `+`. It works by adding the second word on the run time data stack to the working register, and then popping off the second word. Popping is done by a post–decrementing read: read the value pointed to, then decrement the pointer. This is equivalent to popping the 2 top numbers, adding them and pushing the result.

These two examples illustrate how partial evaluation is implemented: by inspecting the compilation stack, the macro `+` knows what code to generate: either the value can be computed at compile time, and the resulting program just quotes the resulting number, or the computation has to be postponed till run time, in which case the appropriate machine instruction is generated.

In the case of the binary operator `+` there is a third possibility: one of its operands might be known at compile time. Starting with a clean compilation stack, providing only one argument yields

```
>> 1 +
        addlw   1
```

which adds the number 1 to the working register, which implements the top of the data stack. The resulting code is still an operation, but it is less general than the one before. The composition `1 +` has been evaluated to a single machine instruction.

4.2 Nested Constructs

Because forth is merely a succession of words, creating nested structures requires some kind of stack. For procedure words nesting, this is the *return stack* which is active at run–time. It records where to continue after terminating the current procedure.

For nested language structures created using macros, this stack is called the *macro stack* and is accessible at compilation time (macro execution time) using the macros `>m` and `m>`. All words that implement nested structures are defined in terms of these two words. For example

```
macro
: begin  sym dup >m label ;
: again  m> jump ;
```

The macro `begin` creates a new symbol, duplicates it and places a copy on the macro stack before creating an assembler label using that symbol. The word again pops the symbol from the macro stack, and uses it to compile a jump instruction. As long as there is a balancing `again` for every `begin`, the resulting code is compilable.

Too many occurences of `begin` lead to non–compilable constructs because the macro stack is not empty. Too many occurences of `again` lead to non–compilable constructs because of macro stack underflow: `m>` will be evaluated without values on the stack.

In the definition of `begin` there is the word `sym`, which creates a new symbol. In an of itself `sym` is not compilable, because the symbol value is not representable on the target system. However, the words `label` and `jump` will consume symbol values to yield constructs that are compilable: assembler labels and jump instructions.

It is legal to use `>m` and `m>` anywhere in macro code as long as the eventual use is balanced. A typical use is in metaprogramming constructs which use a literal value multiple times. For example, a macro that converts a number to a two byte value can be written as

```
macro
: lohi    \ number -- low high
    dup >m
      #xFF and
    m>
      8 >>> ;
```

This will take a literal value, duplicate it and put one copy on the macro stack. The low byte literal is computed by applying a bitmask. The high byte literal is computed by retreiving the value from the macro stack, and shifting its bits to the right by 8. Note that the shift operator `>>>` is only defined at compile time and is thus an ephemeral macro. If the macro `lohi` occurs in a code composition after a computation that yields a literal value, the composition is compilable. The computation runs at compile time so the intermediate results use infinite precision: there is no 8–bit limit for data representation.

Direct access to the compilation stack is convenient, but not always necessary. In the example above `swap` could be used just as well. The following code is equivalent since all its components can be evaluated at compile time.

```
macro
: lohi
```

```
    dup #xFF and swap 8 >>> ;
```

5 Language Features

This section deals with Purrr features that are significantly different from the classic Forth approach.

5.1 Symbols and Quoting

The word `'` a.k.a. *quote* will quote as a literal value the word that follows it. Quoting prevents a symbol from attaining its default behaviour to be compiled. Compilation means to be expanded to some inlined words if it is a macro, or to be transferred into a `cw` (call word) pseudo–machine instruction if it is not.

Symbols can be used as references to on chip data or procedure code, or macros. I.e. the words `@` and `!` interpret literal symbols as variable names, the word `execute` interprets a symbol as a procedure word, and the word `compile` interprets a symbol as a macro, which it will expand, or procedure word.

The word `label` is special in that it allows to *name* the code that will be compiled after its invokation. It is the only way to introduce procedure words, and is called by any word that marks *entry points* into a code sequence. I.e. it is used in the word `:` which defines a procedure entry point: the sequence `: foo` is equivalent to `' foo label`. Such a construct is called a *parsing word*. To stick to Forth syntax for the most part, Purrr contains some words that behave similar to `'`.

Another word that marks entry points is `then`. Code will continue there from a matching `if` word if the condition is false. The word `then` expands to `m> label`, which means it pops a symbol off the macro stack, and uses it to compile an entry point. Looking at the definition of the macro `if`.

```
: if  sym dup >m or-jump ;
```

The word `sym` produces a unique symbol to be used as a name for the code *past* the `if .. then` construct. The macro `if` stores a copy of this symbol on the macro stack, to be consumed by `then` later, and it invokes the `or-jump` macro, which compiles a conditional jump to the label it gets passed.

5.2 Booleans

In Purrr all predicates are macros that can be optimized into efficient machine language conditional branch and skip instructions. By convention macros that produce boolean values are postfixed by a question mark `?` character. The macro `if` can consume these ephemeral boolean values and generate the appropriate conditional jump instruction. Take a (simplified) example from `serial.f` the serial port driver

```
macro
: rx-ready? \ -- ?
    PIR RCIF high? ;

forth
: receive   \ -- byte
    begin rx-ready? until
    RCREG @ ;
```

The macro `rx-ready?` generates an ephemeral boolean derived from the bit at position RCIF (ReCeive Interrupt Flag) in the special function register PIR (Peripheral Interrupt Register). This boolean is consumed by the `until` macro word, which is eventually implemented in terms of the `if` macro word (which itself is implemented in terms of the primitive `or-jump` word). This code illustrates a pattern often used in the Purrr code: abstract each condition in a macro, naming it appropriately to make the code that uses the condition more transparent.

5.3 Tail Call Optimization

In Purrr a procedure word followed by the `;` or `exit` instruction is translated into a jump. This allows for the use of *recursion* to write loops, without overflowing the return stack. The following code does the same as `receive` in the previous example by calling itself recursively until the condition becomes true. This example has multiple exit points (see below).

```
: receive
    rx-ready?
      not if receive ; then
    RCREG @ ;
```

5.4 Predicates for Inspection

Purrr contains a collection of predicates that will just load a flag on the stack, instead of consuming a couple of arguments. This contrasts with some standard Forth predicates. These predicates are named by appending a question mark `?` to the standard Forth name. For example:

```
=   \ a b -- ?
=?  \ a b -- a b ?
```

5.5 Indirect memory access

The PIC18 architecture has separate instruction and data memory spaces. Purr18 uses two pointer registers to access these memories: the `a` register accesses volatile RAM, and the `f` register accesses non-volatile Flash memory. Indirect addressing using the `@` and `!` words is only supported for variables, which are literal addresses. However, it is possible to implement single–byte indirect addressing using the pointer registers.

To read from RAM memory, the words `@a`, `@a+`, `@+a` and `@a-` can be used to access the 4 addressing modes on the PIC18: indirect, postincrement, preincrement and postdecrement. The `a` register can be accessed through the low and high bytes `al` and `ah`. An abbreviation for storing both high and low words is provided:

```
: a!!   \ lo hi --   | store 2 bytes in the a register
    ah ! al ! ;
```

Similarly, to read Flash memory, the words `@f`, `@f+`, `@+f` and `@f-` can be used. The `f` register can be accessed similarly through the byte parts `fl` and `fh`.

5.6 Named Macro Arguments

For macros only, it is possible to give names to literal arguments. For example the word `2@` which fetches 2 bytes from consecutive memory locations can be implemented as

```
macro
: 2@ var |
   var @
   var 1 + @ ;
forth
```

The word `|` is unlike any other word in that it is a parser extension that separates a named argument list from a macro body in which these names might occur. The code above is equivalent to

```
macro
: 2@  \ var --
   dup >m @
       m> 1 + @ ;
forth
```

Note that the occurence of `var` here is just as part of a comment.

6 Control Flow

This sections deals with ways to escape from sequential code execution. The unifying idea is that you can use two stacks to roll your own control abstractions. The *return stack* is used to record nesting state at run time, and for computed jumps (push an address on the return stack, and return or `;` or `exit` to it). The *macro stack* is used for compile time computation of control flow.

6.1 Standard Control Flow Words

Purrr supports the standard control words which use the macro stack. The effect on this stack is indicated as here as `m:` $<$ in $>$ `--` $<$ out $>$. Similarly using `x:` for the *auxiliary stack*.

```
if     \ ? --      m: -- label
else   \ --        m: l0 -- l1
then   \ --        m: label --

for    \ count --  m: -- label   x: -- loopcount
next   \ --        m: label --   x: loopcount --

begin  \ --        m: -- label
again  \ --        m: label --
until  \ ? --      m: label --
while  \ ? --      m: l0 -- l0 l1
repeat \ --        m: l0 l1 --
```

For `for .. next` this is a simplification[3]. Refer to any Forth manual for the meaning of these words.

6.2 Multiple Entry and Exit Points

Since Purrr procedure words are just assembler labels representing machine code addresses, and straight line Purrr code is translated to straight line machine code, there is no reason for a word not to have multiple entry points. In fact, this can be quite convenient. This code

```
: double-increment
    1 +
: increment
    1 + ;
```

defines two words. The second one increments the top of stack value by one, while the first one increments the top of stack value by two. The code in the first definition just *falls trough* to the last definition as if the sequence "`: increment`" wasn't there. Similarly, a procedure word can have multiple exit points. In the code

```
: safe-turn-on
    problem? if ; then turn-on ;
```

[3]What is indicated to be a label is not a symbol bot an anonymous macro representing a backtracking operation used for optimizing loops. It behaves as if it were a label though, it's just impossible to balance them with the other control words

the word `turn-on` is executed if the `problem?` condition is false. If the condition is true however, the word exists trough the `;` word inbetween `if` and `then`.

6.3 Vectors

A *vector* is a variable containing a word address. The interface consists of two words

```
invoke   \ var --     | execute the code stored in var
->       \ var --     | set var to point to code
```

The word `invoke` is implemented as `2@ execute/b`, which fetches 2 bytes from a double variable and passes them to the `execute/b` word which executes the code pointed to using byte addressing. The word `->` stores the address of the code following it in the variable, and then *exits* the word in which it occurs. So it will not execute the code after the arrow, jus change the value of the variable. I.e. the code

```
2variable current-op
: will-inc   current-op -> 1 + ;
: will-dec   current-op -> 1 - ;
```

defines a word `will-inc` that when executed changes the subsequent behaviour of `current-op invoke` to `1 +`. Similarly, the word `will-dec` changes the subsequent behaviour to `1 -`. By itself, the words `will-inc` and `will-dec` don't do anything except for setting the *value* of the variable `current-op`: the word `->` is an *exit point* for these words. We call words containing `->` *arrow words.*

As the name in the example indicates, vectors can be used to set *current* behaviour, folling the Forth mantra "Don't set a flag, set behaviour."

6.4 State Machines

The `route` word is a different mechanism for implementing dynamic behaviour. It can be used to construct *byte codes* using dispatch tables. While vectors are generic because they can point to arbitrary code, byte codes are more specific: they map state (a number) to behaviour by using explicit *interpretation.*

Vectors work well if there is a small number of invokation points and a large number of arrow words. When the number of alternatives is fixed, i.e. in the implementation of *finite state machines*, byte codes are easier to use. The use of `route` is best illustrated with an example of how it would appear in code:

```
: abcd \ bytecode --
    route
        aaa ; bbb ; ccc ; ddd ;
```

In this example, `0 abcd` corresponds to `aaa`, `1 abcd` corresponds to `bbb`, etc. . . This works because for a procedure `aaa`, the sequence `aaa ;` consists of a single machine instruction, and the word `route` takes its argument and uses it to jump past that number of machine instructions. To keep things simple it is best to stick to this kind of behaviour: using procedure words separated by `;` to create a dispatch table.

However, every *slot* in the jump table can be filled with anything that produces a single instruction. You can use macros like `reset` which will compile to the machine RESET instruction, or you can use just `;` by itself, which compiles RETURN and effectively does nothing. It is also possible to leave out the `;` to just jump into a sequence of words skipping the firsts couple.

6.5 Cooperative Multitasking

The heavier approach to sequencing is to use *tasks*. State machines can be the right solution for some problems that do not require *recursion*. When procedure nesting is required but a piece of code does have some control state in isolation of other code, tasks are a good solution. A task has a separate *execution thread*.

Each task's state consists of a set of stacks, in case of Purrr, a return stack, a data stack and an auxiliary stack. Usually it is a good idea to also save a separate copy of the `a` and `f` registers per tasks. Purrr contains primitives to implement your own multitasker in the file `pic18/task.f`. It implements the words `suspend`, `resume` and `swaptask`.

```
suspend  \ -- task       | freeze current task context
resume   \ task --       | make task context current
swaptask \ task var --   | swap task with the task in var
```

Usually the word that performs task switching is called `yield`. In the common case where there are only two separate tasks, this word simply switches between the two tasks, using a single variable to point to the representation of the *other* task:

```
variable other
: yield
    suspend
    other swaptask
    resume ;
```

The difficulty in using tasks on a low level is how to create them. In Purrr this requires manually allocating resources for the tasks's stacks. For the 2–task case the `other` task can be booted by the word

```
: start-other-task
    suspend other !   \ suspend current task
    #x10 rp !         \ use half of the hardware return stack
    #x50 xp !         \ for rp, and use a region of RAM for
    #x60 dp !         \ the byte stacks dp, xp
    task-init-code ;  \ start the task's code body
```

This discards anything stored in the `other` variable, and performs manual context switching by changing the 3 stack pointers to a free memory location, before running the task's initialization code.

A more complicated scheduler can be implemented by replacing the code betwee `suspend` and `resume` in the `yield` code above. For example, code from `pic18/buffer.f` could be used to create a *round-robin* scheduler which executes a couple of tasks in a circular fashon[4].

6.6 Procedures or Macros?

At several points during the development of reusable library code I ran into the question: am I going to use macros or procedure words. To answer the question generally, it should be translated to: should this code be *fast* or *small*.

[4]For PIC18, the hardware return stack is a fairly limited resource. If a lot of tasks are required, explicit copying of the stack might be necessary. An alternative is to write a VM on top of Purrr which doesn't use the hardware stack. There is a draft version of a 16–bit direct threaded interpreter available.

To understand the main reason why this question pops up it is necessary to look at the PIC architecture, where indirect addressing is quite expensive. It is obvious that the PIC has a bias toward *static* objects: it has quite some provisions to deal with memory addresses that are known at compile time, so they can be inlined in the code. However, dynamic access which is necessary for object abstractions requires the use of the FSR registers. Of these there are 3, and Purrr uses them as data stack pointer, auxiliary stack pointer and the `a` register. Whenever an indirect access occurs, the `a` register needs to be saved, set and restored[5]. As a consequence, dynamic objects are about an order of magnitute more expensive than static ones.

This bias toward static code eventually reflects in the design of the Purrr18 library code: it has a lot of provisions for static objects in the form of macros, especially at points where speed might be an issue, for example the buffer code in `pic18/buffer.f`. These are somewhat harder to use because they often need to be *instantiated* explicitly if code size is an issue.

7 Effective 8–bit Programming

This part is specific to the 8–bit Purrr variants, of which there is only one at the moment: Purrr18. Nothing limits Purrr to be implemented on larger word sizes. However, Purrr is organized in a way to make 8–bit data cells practical, while retaining a larger (machine specific) return stack size.

The ANS Standard explicitly prohibits an 8–bit cell size, setting the minimum size at 16 bits. It requires data stack elements, return stack elements, addresses, execution tokens, flags, and integers to be one cell wide. While Purrr is non–standard for a lot of different reasons, this requirement really kills it. However, it is my opinion that an 8–bit Forth has a reason of existence, despite the limitations of different code and data cell sizes.

Purrr contains some 16–bit library routines, but using them can be cumbersome. The Brood distribution contains a direct threaded 16–bit virtual machine written on top of Purrr which does enable a more standard Forth approach. It comes with its own interaction system, similar to Purrr's. This language is substantially different from Purrr and is more true to standard Forth practice. It is however still incomplete.

[5]To get rid of save and restore it is possible to assume throughout the program that the register can get clobbered. However, this is a global constraint which makes it harder to enforce.

In Purrr for the PIC18 the 21 bit wide hardware return stack is used. Purrr only uses the low 16 bits, leading to a representation of a procedure word as a two cell value. Because of its larger size and fixed depth (only 31 words), a separate byte stack called the `x` stack or auxilary stack is used. I.e. this stack is used to store the loop counter in `for` ...`next` loops. It can be used as an alternative to the return stack for temporary value storage.

Working with 8 bit words effectively is all all about sufficiently factored abstract representations, and hierarchical management of large quantities of space and time. The problem points can be identified as limited precision for mathematical operations, limited practical data buffer sizes, limited loop size, and difficulty of representing code as data.

For math, you're basically out of luck and need to resort to tricks. Purrr has some 16–bit math routines, but math–intensive applications usually work better on larger word size (real or virtual) machines, and as such are not considered part of the application domain of straight Purrr code. Building a VM on top of Purrr is the way to go here.

On the other hand, don't forget that logic is your friend! A lot of problems can be solved by creatively using `and`, `or`, `xor`, `-`, `+` and the shift and rotate operations together with the carry flag. Purrr exposes the these low level machine details to give you the means to create your own abstractions on top of them, using either procedure or macro words. Note that hexadecimal numbers are specified like `#xF0`, and decimal numbers like `#x11110000`. Purrr does not use a `base` word: all numbers are decimal, unless they are indicated as hexadecimal or binary. Also note that the PIC18 contains a hardware multiplier for $8 \times 8 \rightarrow 16$ unsigned multiplication, which can be used to build your own multiplication abstraction. Purrr18 contains some code for a 24 unsigned MAC operation to implement digital filters.

The problems caused by large data buffer sizes can usually be avoided by proper abstraction. In addition the intended target chips usually have small memory sizes, so large buffers are rare. When they do occur, it is usually easier to perform buffer management on a byte and a block level: adding hierarchy to a solution can often not only solve a word size problem, but also bring up solutions that are easier to write down. The same argument goes for limited loop sizes. If you need a `for` ...`next` loop that executes more than 256 times, just nest two of them. Even better: put the inner loop in a separate word and try to see if the code now tells you why you're better off using this hierarchical solution in the first place.

For the problem of effectively representing code as data, byte codes bring a simple solution. A byte code can represent up to 256 different words. The easiest way to do this is to use the word `route` to construct a jump table. Here is a code fragment from the boot interpreter taken from the `pic18/interpreter.f` file. It interprets numbers (tokens) ranging from 0 to 15 by mapping them to code.

```
: interpret \ token --
    #x0F and route
              ; receive ; transmit ; jsr    ;
        lda     ; ldf     ; ack      ; reset
        n@a+    ; n@f+    ; n!a+     ; n!f+   ;
        chkblk  ; preply  ; ferase   ; fprog  ;
```

The `route` here is used to perform something akin to procedure table lookup. The word takes a single argument n and jumps to the n–th machine word following itself. The table above contains 16 machine word entries. To make sure jumps remain inside the table, before `route` the top 4 bits of the token are chopped off using `and`. All words in the table, except the empty one and `reset`, revert to procedure words, for which which the idiom `receive ;` compiles to a single machine word jump instruction.

The first slot is empty: a ; word by itself compiles to the RETURN instruction, which in a route table acts as a no–op. The `reset` word is a macro that compiles to the RESET instruction, also taking a single machine word slot.

8 Live Interaction

Next to macro and procedure words, there is a class of words only defined for live interaction. Two of these interactive commands prints out documentation for words.

```
see  <WORD>
msee <MACRO>
```

it is possible to inspect the target or macro code associated to a certain word. A procedure word is always compiled machine code: all information about the source code has been lost.

```
> see receive

receive:
        btfss  158, 5, 0
 (106)  bra    receive
 (108)  btfss  171, 1, 0
 (110)  bra    118
 (112)  bcf    171, 4, 0
 (114)  bsf    171, 4, 0
 (116)  bra    receive
 (118)  movwf  236, 0
 (120)  movf   174, 0, 0
 (122)  btfss  171, 2, 0
 (124)  bra    130
 (126)  movf   237, 0, 0
 (128)  bra    receive
 (130)  return 0
```

A macro word is either a composition of other macros,

```
> msee begin
macro:
(sym dup >m label)
```

a primitive assembler pattern matching macro,

```
> msee +
asm-match:
((((qw a) (qw b) +)       ((qw (wrap: a b '+))))
 (((addlw a) (qw b) +)    ((addlw (wrap: a b '+))))
 (((qw a) +)              ((addlw a)))
 (((save) (movf a 0 0) +) ((addwf a 0 0)))
 ((+)                     ((addwf 'POSTDEC0 0 0))))
```

or one of the few CAT primitives with `state:` syntax.

```
> msee m-dup
macro-prim:
(dup)
```

References

[1] Purrr, 2007.

[2] Kelsey, R. Pre-scheme: A scheme dialect for systems programming.

[3] Schouten, T. Brood, 2007.