Metaprogramming for the Masses

Alaric B. Williams (alaric@alaric-williams.com)

Imperial College, London

June, 2000

Abstract

In this report, the often-ignored topic of language preprocessing systems, or macro languages, is revisited from the viewpoint of *metaprogramming* and a preprocessor/macro system capable of subsuming many of the front-end stages of compilation developed. The far-reaching consequences of this for the design and usage of programming languages are examined in turn, with the help of a simple toy language implementing little else but the author's preprocessing system but which proves itself capable of remarkable expressive power. Notes for the application of the system to "real" languages are given in the text.

Metaprogramming for the Masses

Alaric B. Williams (alaric@alaric-williams.com)

Imperial College, London

June, 2000

1. Introduction

Much research has been put into the holy grail of increasing the expressive power of programming languages. Although some would argue that any Turing-complete language is equally "expressive", most will point out that they'd rather write compilers in a language with pattern matching than in raw machine code.

Developments of note in the field include Knuth's work on source documentation and control structures (including the controversial *GOTO* construct) [4], various works on object orientation (a technique for bringing programming closer to the problem domain), studies of novel ways to express imperative operations [2], the study of functional languages, and research into languages intended for problem domains where the normal applicative structure we have grown accustomed to becomes a hindrance, and more esoteric technologies arise. These include the logical [6, section 4.4] and nondeterministic languages [6, section 4.3].

One area, however, which has traditionally been neglected, is the study of language constructs dealt with before the compiler gets to see the source. Typically, a seperate *preprocessor* accepts a source file, and produces one with identical semantics but expressed in a subset of the original language - the subset made by removing the *preprocessor constructs* which the preprocessor acts upon from the language.

Many of the early stages of "real" compilation in modern languages consist of reducing the result of the last stage into a semantically identical program in a simpler language. The Standard ML compiler, for example, simplifies programs into a continuation passing language, then proceeds to simplify it into continuation passing languages that are closer and closer to assembly language before finally handing over to what I would actually call a compiler, which produces machine code [1].

However, those compiler stages are not implemented in the preprocessor due to lack of expressiveness of the preprocessor sublanguage itself. For example, in C or C++, the preprocessor allows the programmer to express simple source transformations at the token level, for example:

```
/* This is a comment, which the preprocessor removes */

/* The next line "includes" source from another file at this
point; this allows us to use the preprocessor to reuse common
pieces of source */
```

```
#include <stdio.h>

/* Here we command the preprocessor to, from this point onwards,
replace the token PI with the token 3.14159

#define PI 3.14159

/* A more interesting example. This commands the preprocessor to,
upon meeting the token DOTPROD, parse a bracketed list of
six arguments, and replace the whole thing with the list of
tokens given after the space, substituting the token strings
that matched each argument, as you might expect. Note the need
to enclose the arguments in brackets, in case they are themselves
arithmetic expressions containing lower precedence operators
than *. The same operation could have been expressed more
neatly as a function, but this type of thing was fairly common
in the days when compilers produced signification overhead
for function calls. */

#define DOTPROD(x1,y1,z1,x2,y2,z2)
((x1)*(x2)+(y1)*(y2)+(z1)*(z2))

/* Here, on the other hand, is something that can only be done
through the preprocessor; we introduce a new language construct
that, in a manner most unfamiliar to the average C programmer,
creates a function. */

#define MAPPER(mappername,elementtype,mapfunc) \
void mappername(elementtype *array,int num_elems) { \
        while ( num_elems --) { \
                *array = mapfunc(*array); \
                array++; \
        } \
}

int square(int x) {
        return x*x;
}

MAPPER(array_square, int, square)

void test_mapper() {
        int my_array[4] = {1,2,3,4};
        array_square(my_array,4);
}
```

The C preprocessor is crippled both by the fact that it can only expressions the triggers of transformations as being the constructs the preprocessor transforms itself - prefixed by a # symbol - or "function-like" constructs like our MAPPER example; and the fact that once it has recognised a programmer-defined transformation, it can do no more than a very simple token substitution.

Scheme has a macro system. However, it also has a very regular syntax in which every programming language construct is a list with the name of the construct at the head and the remaining elements interpreted in a manner dependent on the construct. For example, a conditional expression is represented as `(if control-variable then-arm else-arm)`, while a lambda expression is written `(lambda (argument1 argument2) body)`.

This allows the macro system to be better than C's in that the new constructs created by programmers can look like any other construct - and vice versa, that any standard programming language construct could actually be implemented by a source transformation rather than requiring special case code in the compiler. The Scheme Report[5] actually presents a simple core language and uses the source transformation system to add higher level language constructs to it.

However, the downside of Scheme's source transformation system is that it still can only perform limited substitution operations, much like the C preprocessor although definitely an improvement. It introduces a pattern matching language to express multiple possible rewritings depending on the body of the input construct, which although useful is another thing for the programmer to learn; and many Scheme programmers don't bother, considering it an advanced esoteric topic.

Perhaps the most technically advanced source transformation system in the literature so far is the FORTH programming language. A FORTH compiler takes input source, represented as a series of whitespace seperated "words", and converts it into a simple stack machine code with the operations `push` and `call`. The runtime system contains a wide range of primitive words to `call`.

The compiler loops through the input source, reading a word at a time. For each word, if the word is a synactically valid number, it emits a push instruction to push that number. If it is a word, it looks it up in the *word table*; if the word is marked as *immediate* it is executed immediately by the compiler, otherwise a call to the word is emitted.

The word table is seeded with various words, perhaps the most important of which is "`:`", which is used to define new words. `:` is an immediate word, which when executed reads a single word from the input stream to be the name of the new word; it then saves the output state of the compiler and creates a new output state pointing to a free memory buffer, and lets the compiler compile into that buffer as usual until another special immediate word, "`;`", is found, at which point the resulting compiled code is put into the word table under the supplied name, and the compilation state is restored. A variant on this is used to create immediate words. A word definition is often written with a comment in round brackets showing the "stack picture", which we would normally think of as the type of the function. For example, a function accepting a byte (character) and returning the same might be:

```
: rev8 (ch -- ch) 0 SWAP
                  8 0 DO DUP 1 AND
                         ROT 2* OR
```

```
                                SWAP 2/
                            LOOP
                        DROP ;
\
\ Brute force bit reversal of a byte.
```

Clearly, a FORTH compiler can be written very easily; the power of the language is almost entirely in the standard libraries. Strings are handled by an immediate word, "`S"`", which reads characters until a closing quote is found, when it stores the result into a memory block and emits code to push the address of the string; this allows us to write literal strings in our source. Comments are *implemented* by the "`(`" word, which discards input until a closing bracket is encountered.

As some people are unfamiliar with the stack machine or reverse polish architecture, they may want to use a FORTH module that implements a recursive decent parser. To create such a beast, we might define an immediate word "`EVAL(`" which parses and compiles an arithmetic expression terminated by a closing bracket. That's a fairly drastic thing to be able to do with an easily written extension library.

Perhaps the ultimate extension of that idea is to write compilers for totally unrelated languages - as immediate FORTH words. A C compiler could be written as an immediate word, allowing you to imbed C source code in your FORTH programs. The beauty of this compilation architecture is only offset by the fact that FORTH is a relatively unweildy and untyped stack language.

FORTH's architecture hasn't arisen in anything other than FORTH derivatives and clones. Certainly, as far as the author has been able to ascertain, nothing like it has never been used in a prefix notation applicative language of the type we are familiar with today. The real crux of the system is that not only can the developer play with language constructs at precisely the same level of power as the constructs built into the core language, the actual transformations applied to the constructs are expressed in the target language itself, rather than a limited substitution system or a totally seperate pattern matching language.

1.1. About this project

The aim of this project is to develop a compilation system reminiscent of FORTH's, but brought up to date with modern language architectures. Sadly, nothing of the techniques used to implement FORTH can be carried far beyond languages with the its unique structure, requiring a total redesign. It will be shown that compilers can be drastically simplified by using "standard libraries" of common language constructs built upon a very simple core language supplemented with a syntactic transformation "preprocessor", instead of dealing with a wide range of constructs in the compiler itself. It will also be shown that the ability to add powerful new constructs easily can be used to massively extend the expressive power of the language.

On the one hand, we can implement speclaised constructs designed for the problem domain our programs are working in; and on the other hand we can increase performance and reliability with syntactic transforms designed to optimise code or perform extensive "precompilation" of abstract high level problem descriptions into the tight, nasty, tangled code common in hand

optimised algorithms. This also decreases the opportunity for bugs to creep in by automating the optimisation process.

Knuth[4] produced the earliest reference known to the author concerning the process of producing optimised code by first writing an implementation emphasising clarity and maintainability, then applying semantic-preserving optimisation transformations to that code one by one until an optimised version is produced. The only difficulty with that approach is maintainability - if the algorithm is to be changed, the change is either applied to the original maintainable form and the optimisations reapplied laboriously, or the optimised version can be patched at the cost of possibly introducing confusing bugs. In this report, a development process is suggested where the high level algorithm is developed then a series of source transformations added to it to show the compiler how to "hand optimise" it; the advantage being that the original code can be changed, and the compiler will still apply the optimising transformations given to it. Of course, changes to the algorithm will often warrent new optimisations and make old ones obsolete or invalid, but an environment in which optimising transformations can be simply commented out to see if they are causing a bug or are not pulling their weight any more should drastically ease this process.

2. The Qube language

We begin by defining a toy language with barely enough primitives to make it Turing-complete, then proceed to augment it with the preprocessor developed in this project. Although the Qube language is not intended for use beyond the research lab, the author gives notes about the issues involved in implementing the source transformation system for a "real" language where applicable.

2.1. The core language

Qube is a very simple nearly-functional language; the author's objective in starting with such a trivial language is both to avoid repeating the well-known steps involved in creating a conventional language, and to make the language simple enough to show that it has no particularly special features to support the preprocessor that is going to be wrapped around it. This makes it clear that the preprocessing technology could just as easily be applied to an existing "real world" compiled language.

2.1.1. Language primitives

- `(fun (<args>) <body>)` creates a function, just like the familiar lambda construct of many functional languages but with a shorter and more layperson-friendly name. For example:

```
(fun (counter increment) (+ increment counter))
```

- `(if <a> <b> <c>)` if a is the boolean false value, evalutes to the value of c, otherwise the value of b. For example:

```
(if (> 21 age) Old Young)
```

- `(<func> <args…>)` applies the function to the arguments, and evaluates to the result of the application. For example:

```
(+ 1 2)
```

- `(extern <string>)` "includes" the contents of the file named in the string. For example:

```
(extern "$QUBELIB/my-print-func.qube")
```

- `(quote <value>)` evaluates to the given value; `quote` acts as a way to prevent lists from being interpreted as function applications, or symbols as variable references. The Scheme parser inherited by Qube treats the string "`'x`" as "`(quote x)`"; this is a syntactic convenience. For example:

```
(quote (this is a list))
(quote a-symbol)
'another-symbol
```

- A `<symbol>` evaluates to the value of the symbol in the current lexical environment (defined by `fun` constructs). For example:

```
current-time
```

- `"<string>"`, `<integer>`, or `<float>` all evaluate to said string, integer, or float; these are literal constants. For example:

```
"String values can contain \"escaped\" quotes"
156
1.3
```

The only construct that binds new variables is `fun`. These bindings have lexical scope, and functions close over their environments, as is the norm for functional languages. The top level environment in which programs are evaluated is preset with a selection of standard functions:

- `(prepend <anything> <list>)` is the direct analogue of Scheme's `cons` function. `prepend` takes an object of any type and a list, and returns a new list with the object at its head and the passed list as the remainder of the list. For example:

```
(prepend 1 '(2 3)) = '(1 2 3)
```

- `(head <list>)` returns the head (first element) of a list. For example:

```
(head '(1 2 3)) = 1
```

- `(tail <`*`list`*`>)` returns the tail of a list, the tail being all the elements of the list after the first; for example,

```
(tail '(1 2 3)) = '(2 3)
```

- `(list? <`*`anything`*`>)` returns the argument if it is a list and returns the empty list `'()` otherwise.
- `(fun? <`*`anything`*`>)`, `(number? <`*`anything`*`>)`, `(string? <`*`anything`*`>)` and `(symbol? <`*`anything`*`>)` all test their argument for membership of the named type, in the same manner as the previous function.

From these functions the usual library of functions in a LISP-like functional language can be derived - map, fold, and friends, which I shall use in later examples without explanation; the interested reader should consult one of the many Scheme reference books[6].

2.2. Partial evaluation

The core of the Qube preprocessor is a partial evaluation engine. Partial evaluation is the process of evaluating an expression when some of its variables are unknown. Whereas in normal evaluation the result is a single value, the result of partial evaluation can either be a value (if there are no unknowns in the expression, or few enough unknowns to make the final result decidable) or an expression in terms of those unknowns. In the worst case, the output expression might be exactly the same as the input expression; however, little code written by humans exhibits that many unknowns, and a large proportion of code can be executed at "compile time" to produce a somewhat simplified program.

The Qube partial evaluation engine consists of rules for partially evaluating each basic form, which usually start by recursively partially evaluating the subforms within to see if they can be reduced to simple values. This allows the rule to select from a number of cases depending on which parameters are fully known and which still contain some unknowns.

Each rule has, as inputs, the expression to be matched and the environment of static bindings it is to be partially evaluated in. The environment is represented as a binary relation mapping symbols to values.

The result of partially evaluating an expression or subexpression is a member of one of three types:

- Static values are constants. A partial evaluation will return a static value if there is no uncertainty present, meaning the expression has a statically decidable value. The notation used for static values is defined as:
  - *STATIC*(x) constructs a static value.
  - *unbox*(*STATIC*(x)) = x since the *unbox* function extracts the value from a "type wrapper".
  - *isStatic*(*STATIC*(x)) is always true.

- Function values are functions which do not close over any environmental bindings that are not reducable to static values at the point of function definition; in other words, the body of the function can be inserted verbatim at the point of function application without fear of namespace clashes. Function values are a special case of static values which allow for partial evaluation of function applications. Function value notation is defined as:
  - $FUNCTION(x)$ constructs a function value.
  - $unbox(FUNCTION(x)) = x$, as you might expect.
  - $isStatic(FUNCTION(x))$ is always true, since functions are static values.

- Dynamic values are the results of partially evaluating expressions that cannot be fully evaluated to a single value, generally due to a reference to an unbound variable in the body of the expression. The actual "value" is just the partial evaluator's best attempt at partially evaluating the expression, and may be amenable to further evaluation when higher level binding constructs have been taken into account. The notation is:
  - $DYNAMIC(x)$ constructs a dynamic value.
  - $unbox(DYNAMIC(x)) = x$, as you might expect.
  - $isStatic(DYNAMIC(x))$ is always false.
  - $isDynamic(x)$ *iff* $\neg isStatic(x)$

The values bound to symbols in an environment are also type-tagged as above, but only *STATIC* and *FUNCTION* types are valid in environments.

For clarity, I write rules in the form

$$environment \vdash expression \mapsto result$$

Symbols with tildes (eg, $\tilde{x}$) on the left hand side are considered to be pattern wildcards which are bound to the values they match on the left hand side when they appear on the right hand side.

I have also adopted the convention of representing the partially evaluated form of some subexpression on the left hand side of the rule with an underlined version of that symbol on the right hand side. However, this symbol is always defined in the text.

I have used a hat over a symbol in a small number of cases to represent a version of the value of the unadorned symbol that has been processed in some complex way, which is also always defined in the text.

2.2.1. Literals

For example, the rule:

$$\tilde{E} \vdash \tilde{i} \mapsto STATIC(\tilde{i}) \text{ iff } \tilde{i} \in integers$$

states that an integer, $\tilde{i}$, in an environment $\tilde{\mathrm{E}}$, partially evaluates to $STATIC(\tilde{i})$. Likewise:

$$\tilde{\mathrm{E}} \vdash \text{"}\tilde{s}\text{"} \mapsto STATIC(\tilde{s}) \ \textit{iff} \ \tilde{s} \in \textit{strings}$$

makes a similar statement about strings, and:

$$\tilde{\mathrm{E}} \vdash \tilde{f} \mapsto STATIC(\tilde{f}) \ \textit{iff} \ \tilde{f} \in \textit{floats}$$

makes the same statement for floating point numbers. A slightly more complex looking rule states that the `quote` construct always partially evaluates to the enclosed value:

$$\tilde{\mathrm{E}} \vdash (\textit{quote} \ \tilde{x}) \mapsto STATIC(\tilde{x})$$

2.2.2. Variables

Symbols, if bound in the environment, reduce to the value they are bound to; this value will already be tagged with STATIC or FUNCTION so no further tagging is required.

$$\tilde{\mathrm{E}} \cup \{\tilde{x} \rightarrow \tilde{y}\} \vdash \tilde{x} \mapsto \tilde{y}$$

$$\tilde{\mathrm{E}} \vdash \tilde{x} \mapsto DYNAMIC(\tilde{x}) \ \textit{otherwise}$$

If a symbol is not defined in the environment, then it is left as a dynamic expression.

2.2.3. Conditionals

Conditionals are, again, slightly more complex. In the case where the control value in the conditional expression is decidable and not equal to the `false` value, we can reduce the conditional expression to the first branch:

$$\tilde{\mathrm{E}} \vdash (\textit{if} \ \tilde{c} \ \tilde{x} \ \tilde{y}) \mapsto peval(\tilde{\mathrm{E}} \vdash \tilde{x}) \ \textit{iff} \ peval(\tilde{\mathrm{E}} \vdash \tilde{c}) \neq STATIC(\textit{false})$$

Likewise, if the control value is decidable and **is** equal to the `false` value, we can reduce the conditional expression to its second branch:

$$\tilde{\mathrm{E}} \vdash (\textit{if} \ \tilde{c} \ \tilde{x} \ \tilde{y}) \mapsto peval(\tilde{\mathrm{E}} \vdash \tilde{y}) \ \textit{iff} \ peval(\tilde{\mathrm{E}} \vdash \tilde{c}) = STATIC(\textit{false})$$

However, if the conditional cannot be statically determined, then the best we can do with the conditional expression is replace the three subexpressions with their partially evaluated forms. Perhaps we could test to see if both arms are equal and, if so, not bother with evaluating the conditional, but I do not consider that to be a common enough case to concern myself with.

$$\tilde{\mathrm{E}} \vdash (\textit{if} \ \tilde{c} \ \tilde{x} \ \tilde{y}) \mapsto DYNAMIC((\textit{if} \ \underline{\tilde{c}} \ \underline{\tilde{x}} \ \underline{\tilde{y}})) \ \textit{iff} \ \underline{\tilde{c}} = STATIC(\textit{false})$$

where

$$\underline{\tilde{c}} = peval(\tilde{\mathrm{E}} \vdash \tilde{c}),$$

$$\underset{\sim}{\tilde{x}} = peval(\tilde{\mathrm{E}} \vdash \tilde{x}),$$
$$\underset{\sim}{\tilde{y}} = peval(\tilde{\mathrm{E}} \vdash \tilde{y})$$

2.2.4. References to external files

The extern construct is what would normally be called "include" in most languages, but that just sounds too verb-like to be in a functional language. Extern is used to refer to source code contained in an external file; if the filename is decideable at preprocessing time, and the file can be found at preprocess time, then it is included there and then.

$$\tilde{\mathrm{E}} \vdash (extern\ \tilde{x}) \mapsto peval(\tilde{\mathrm{E}} \vdash e)\ iff\ \underset{\sim}{\tilde{x}} = STATIC(filename) \wedge FileExists(filename)$$

where

$$\underset{\sim}{\tilde{x}} = peval(\tilde{\mathrm{E}} \vdash \tilde{x}),$$
$$e = FileContents(filename)$$

However, otherwise, all we can do is partially evaluate the filename expression.

$$\tilde{\mathrm{E}} \vdash (extern\ \tilde{x}) \mapsto (extern\ \underset{\sim}{\tilde{x}})\ otherwise$$

2.2.5. Functions

The introduction of functions with the `fun` construct will fall into one of two cases.

The body of the function is partially evaluated in a special environment, where the arguments of the function have been bound to a special static PLACEHOLDER value; if $\hat{e}$, the result of this partial evaluation, is static then the function body has no free variables other than arguments to the function, or that all the non-free variables it refers to can be statically reduced to constant values and thus eliminated. This condition MUST hold before we can safely lift the function body away from its point of definition to its point of use; and so, if it holds, we return a FUNCTION value.

$$\tilde{\mathrm{E}} \vdash (fun\ \tilde{a}\ \tilde{e}) \mapsto FUNCTION(\tilde{a}, unbox(\underset{\sim}{\tilde{e}}))\ iff\ isStatic(\hat{e}),$$

where

$$\underset{\sim}{\tilde{e}} = peval(\tilde{\mathrm{E}} \vdash \tilde{e}),$$
$$\hat{e} = peval(\mathrm{E} \oplus \{x \rightarrow STATIC(PLACEHOLDER) | \forall x \in \tilde{a}\} \vdash \tilde{e})$$

If the body is not safely liftable, it has to be left as is and dealt with at run time or when higher level rewritings provides values for all the variables referenced in the function body.

$$\tilde{\mathrm{E}} \vdash (fun\ \tilde{a}\ \tilde{e}) \mapsto DYNAMIC((fun\ \tilde{a}\ \underset{\sim}{\tilde{e}}))\ otherwise$$

2.2.6. Function application

Function applications can only be partially evaluated to a simpler form if all of the arguments being passed to the function are statically decidable. The reason for this is that dynamic

argument expressions will need to be evaluated in the correct environment; if they are substituted into the body of the function then variables they refer to may have been rebound to other values. This can be fixed by wrapping them up in closures and various such techniques, but this level of complexity is not necessary here.

$$\tilde{\mathrm{E}} \vdash (\tilde{f}\ \tilde{a}\ \ldots) \mapsto \mathit{DYNAMIC}((\underline{\tilde{f}}\ \underline{\tilde{a}} \ldots))$$
$$\mathit{iff}\ \mathit{isDynamic}(\underline{\tilde{f}}) \vee \mathit{isDynamic}(\underline{\tilde{a}}) \vee \ldots$$

where

$$\underline{\tilde{f}} = \mathit{peval}(\tilde{\mathrm{E}} \vdash \tilde{f}) \ldots$$

If all of the arguments are static values and the function to be applied is a FUNCTION value, we may be able to perform an in-place beta expansion, or *inlining* as it is usually known:

$$\tilde{\mathrm{E}} \vdash (\tilde{f}\ \tilde{a}\ \tilde{b}\ \ldots) \mapsto \mathit{peval}(\hat{\mathrm{E}} \vdash \hat{e})\ \mathit{iff}\ \underline{\tilde{f}} = \mathit{FUNCTION}(\hat{a}\ \hat{b}\ \ldots, \hat{e})$$

where

$$\underline{\tilde{f}} = \mathit{peval}(\tilde{\mathrm{E}} \vdash \tilde{f}),$$
$$\hat{\mathrm{E}} = \{\hat{a} \rightarrow \underline{\tilde{a}}, \hat{b} \rightarrow \underline{\tilde{b}}, \ldots\}$$
$$\underline{\tilde{a}} = \mathit{peval}(\tilde{\mathrm{E}} \vdash \tilde{a}),$$
$$\underline{\tilde{b}} = \mathit{peval}(\tilde{\mathrm{E}} \vdash \tilde{b}) \ldots$$

Note how the environment to partially evaluate the body of the function in is generated by binding the names of the function's arguments to the values supplied by the caller, which are all static values since the first rule for function applications did not match.

If the function is a STATIC value, then there can be one of two cases; either it's the PLACEHOLDER value, indicating a function argument that is not available presently but will be decidable when the function is applied, or the function will be an actual function object (as opposed to a reference to a function introduced by the `fun` construct, which would be tagged as FUNCTION instead of STATIC); these only come into the system through the initial top level environment, which is set up to contain the primitive function library described previously.

In the former case we can save some time by just reducing the function application to the PLACEHOLDER value. In the latter case the function can be applied there and then, making it possible to compute the value of expressions in the preprocessor, at "compiler time" (instead of every time an enclosing loop is executed!). The rule handling this can be written:

$$\tilde{\mathrm{E}} \vdash (\tilde{f}\ \tilde{a}\ \tilde{b} \ldots) \mapsto \mathit{STATIC}(\mathit{apply}(\hat{f},\ \underline{\tilde{a}},\ \underline{\tilde{b}},\ \ldots))$$
$$\mathit{iff}\ \underline{\tilde{f}} = \mathit{STATIC}(\hat{f}) \wedge \hat{f} \neq \mathit{PLACEHOLDER}$$

$$\tilde{\mathrm{E}} \vdash (\tilde{f}\ \tilde{a}\ \tilde{b} \ldots) \mapsto \mathit{STATIC}(\mathit{PLACEHOLDER})$$
$$\mathit{iff}\ \underline{\tilde{f}} = \mathit{STATIC}(\mathit{PLACEHOLDER}) \vee \underline{\tilde{a}} = \mathit{STATIC}(\mathit{PLACEHOLDER}) \vee \underline{\tilde{b}} = \ldots$$

These rules must be applied iteratively, starting with the input source expression until a fixed point is reached. Together they make up the $\mathit{peval}(\mathrm{E} \vdash x)$ function, which will reduce any Qube source expression to a significantly simpler (though not necessarily smaller) form.

However, this is nothing new in itself.

For example, the input expression `(f a b)` in the environment $\{f \rightarrow FUNCTION(x\ y, (\texttt{PrimitiveAdditionOperation}\ x\ y)),\ a \rightarrow 1\}$ will initially match one of the function application rules. Therefore, the three subexpressions are partially evaluated in turn; the partial evaluation of `f`, through the variable reference rule, returns the function from our environment and the partial evaluation of `a` yields the static integer 1. However, `b` matches the rule for variables not bound in the enclosing environment so will only reduce to $DYNAMIC(b)$; this means that the rule for a function application where not all of the subexpressions are static is invoked, making the result $DYNAMIC((f\ a\ b))$.

However, should `b` have been bound to the integer `2` in the environment, a different result would emerge. All of the subexpressions of the application would have reduced to static values, with the first being a *FUNCTION* value, thus permitting the use of the inlining rule. This would create an environment for the body of the function like so:

$$\{x \rightarrow 1,\ y \rightarrow 2\}$$

The body of the function would be partially evaluated in this environment. The body is:

$$(\texttt{PrimitiveAdditionOperation}\ x\ y)$$

The first element, being already of function type, will partially evaluate to itself, while `x` and `y` partially evalute to `1` and `2` respectively from the environment. The result is an expression matching the rule for a function application where the function is a primitive, which is evaluated by applying the function, in this case yielding the integer `3`; this, marked as a static value, can then be the result of the entire partial evalation:

$$STATIC(3)$$

2.3. The `meta` construct

The preprocessor also adds an additional construct to the language, called `meta`. This construct is remarkably simple to define, yet has a very profound effect on the expressive power of the language.

The rule for partially evaluating the `meta` construct is:

$$\tilde{\mathrm{E}} \vdash (meta\ \tilde{x}) \mapsto peval(\tilde{\mathrm{E}} \vdash unbox(\underline{\tilde{x}}))$$
$$iff\ \ isStatic(\underline{\tilde{x}})$$
$$where\ \underline{\tilde{x}} = peval(\tilde{\mathrm{E}} \vdash \tilde{x})$$

In the ideal case, the `meta` construct fully evaluates its enclosed subexpression to a static value. It then treats this as an expression, partially evaluates it, and returns that as the result of its own partial evaluation. In other words, the `meta` construct allows for metaprogramming; instead of just executing the code inside the `meta`, the value resulting from executing said code is executed.

In the event of the body of a `meta` construct being a placeholder or a dynamic value, we

cannot properly evluate it yet, and have to defer the process appropriately:

$$\tilde{\tilde{E}} \vdash (meta\ \tilde{x}) \mapsto STATIC(PLACEHOLDER)$$
$$iff\ \tilde{\underline{x}} = STATIC(PLACEHOLDER)$$
$$\tilde{\tilde{E}} \vdash (meta\ \tilde{x}) \mapsto DYNAMIC((meta\ \tilde{\underline{x}}))\ otherwise$$

3. Programming with the `meta` construct

Perhaps the most obvious use of the `meta` construct is to the introduction of new language constructs. If we define a compiler function that returns valid Qube source for an expression with semantics somehow controlled by the function's arguments, we can use it like so:

```
(meta (compile-let '(
        (x (+ 1 a))
        (y (+ c d)))
        '(+ x y)))
```

Where the `compile-let` function might be defined as:

```
(fun (bindings body)
        (prepend (prepend 'fun
                 (prepend (map head bindings)
                 (prepend body '())))
        (prepend (map (fun (x) (head (tail x))))
                 '())))
```

The partial evaluation of the above `meta` expression starts by partially evaluating the enclosed expression, in this case the application of `compile-let` to some arguments. `compile-let` simply constructs a `fun` expression with the variables bound in the virtual `let` expression as arguments, with the body of the expression passed through, then wraps this function expression in an application which binds the variables to their values. In this case, `compile-let` would return the static value:

```
((fun (x y) (+ x y)) (+ 1 a) (+ c d))
```

This is where the magic occurs. The `meta` construct then partially evaluates this *again* and then uses the result of the second partial evaluation as the result of partially evaluating the entire `meta` expression.

In other words, a function applied at *compile* time has returned a snippet of source code which is then used in the program. To put it another way, the programmer has been able to write code that writes his or program!

3.1. Implementing a module system

The previous example was an ugly way to express `let` statements. While Scheme's macro system would require you to use a special rewrite rule language to express `let` in terms of `lambda`, at least the result wouldn't need all that quoting, and it would be `(let … )` instead of `(meta (compile-let … ))`. However, this is easily fixed, along with several other problems; using the `meta` construct, we can turn this toy language, in which programs are nothing more than expressions to be evaluated, into a "real" language with modules and namespaces! All that is required is to implement a top level compiler function that provides a syntax for a "module" with exported names, imports from other modules, and a way of setting up more aesthetically pleasing ways to invoke metaprogramming constructs like compiler functions.

The first tool in my metaprogramming library was the module system. A "module" is represented as a list containing:

1. A list of pairs of the form `(name value)`, which represent names exported by the module
2. A list of functions which are to be used as source transformations; all expressions in the module importing this module are passed through each function in this list in turn. This is explained in more detail below.
3. A list of metaprogramming constructs, represented as pairs of the form `(name compiler-func)`. This is just a special case of the previous element; if the form `(name … )` is found in the body of code importing the module, then the form is replaced by the result of applying the `compiler-func` to the form. This makes it easy to add new constructs like `let` to the language.

A Qube module file is of the form:

```
(meta ((extern "$QUBELIB/module.qube")
(module "modulename")
(import (module1 name1) (module2) …)
```

Then any number of each of these statements, in any order:

```
(private name expression )
(public name expression )
(transform expression )
(construct name expression )
```

Terminated by:

```
))
```

The `import` declaration lists modules that are imported into the namespace of this module, along with the name the module is being referred to, which is prefixed to all binding and construct names imported from the module if specified, seperated by a `.` (dot) character. This allows the user to import from two different modules that export the same public names.

Normally, modules for importing would be obtained with an `(extern "filename")`

construct referring to a file containing a Qube module definition.

The `private` and `public` declarations in the module body are combined together into a single recursive `let` expression, which is in turn implemented with a `fun` using a form of fixpoint combinator, to allow the values specified in any of the four kinds of declaration to refer to both public and private bindings in the module. The "value" of the resulting expression is nothing more than the module implemented as a list as above.

All of the expressions in the declarations are passed through the `transform` functions provided by any imported functions, in the order in which they appear, then scanned for references to construct names defined in imported modules, optionally prefixed if a name was given to the import. The code that performs this transformation is defined as a function in `"$QUBELIB/module.qube"`; so if a user wishes to change the module system for whatever reason, they can trivially create a new one, perhaps compatible with this one's representation for modules if not the syntax used.

Having implemented the module system, a library of standard constructs can then be implemented such as `let` and `cond`. While doing this in his sample implementation, it became clear that a method was needed to combine several modules into one.

For example, a series of modules providing standard functionality in various areas could be combined into a single "common functionality" module. It was surprisingly easy to write a function which created a "library" module composed from a series of modules:

```
;; Usage:  (make-library (<module list…>))
(public make-library (fun (modules)
        (let (
                (bindings (append-list
                        (map (fun (x) (elem 1 x)) modules)))
                (processors (append-list
                        (map (fun (x) (elem 2 x)) modules)))
                (constructs (append-list (map
                        (fun (x) (elem 3 x)) modules)))
                )
        (list bindings processors constructs))))
```

The workings of this function are simple: the `elem <n> <list>` function returns the n'th element of the list, and is used with `map` to prepare lists of all the bindings, processors, and constructs in the passed list of modules; each of these lists of lists is then merged into one list with the `append-list` function, which "flattens" out one level of nested lists. This results in a set of three lists which can be considered the union of the corresponding lists within the input modules; if bindings or constructs in any of the modules have naming conflicts with others, the conflict resolution in the module system itself will simply give precedence to the "rightmost" definition, so we do not need to remove duplicates when constructing libraries. These three lists can then be used to construct a new module object.

This allows a standard library for Qube modules to be created, and imported with `(extern "$QUBELIB/core.lib")`. The unpleasant syntax for introducing modules (`(meta ((extern "$QUBELIB… ))))`) can also be replaced with a much nicer system; the file `module.qube`

containing the module system can be extended to define a top level module, which differs from a normal module in that it cannot contain `public` declarations, and instead contains a single `main` declaration. The top level module is transformed into a recursive let-like statement as per the normal module construct, but instead of evaluating to a module object, it instead evaluates to the result of evaluating its `main` declarataion. This module also automatically imports the core library, which was extended to support a new `module` construct which can be used to create normal modules with the enhanced syntax:

```
(module "modulename"
(import (module1 name1) (module2) …)
```

Then any number of each of these statements, in any order:

```
(private name expression )
(public name expression )
(transform expression )
(construct name expression )
```

Terminated by:

```
)
```

Having moved the initial use of the `meta` construct to extend basic Qube into a serious programming language into a standard library, we can now make a wrapper script to take a file comprising nothing other than the body of a top level module, then transparently wrap it between:

```
(meta ((extern "$QUBELIB/module.qube")
```

and

```
))
```

With relatively little work, we have now extended the primitive Qube language almost beyond recognition!

3.2. The implications of `meta` for compiler design

The ease with which it is possible to bolt a fully featured module system onto a language as primitive as Qube using our `meta` construct comes as quite a surprise. The only features of the underlying language beyond `meta` that were essential in this were its functional purity, which makes it safe to rearrange code with `meta` without having to worry about side effects, and the `extern` construct which provides a simple way to bring code in from the filesystem. Neither of these things are particularly special; the magic is all in the partial evaluation and the `meta` construct added by the preprocessor.

As we make the language richer and richer with high level constructs, the bulk of the language's complexity quickly moves into the standard library, making the compiler itself seem more and more like nothing more than an assembler. In effect, we have created a type of "open

compiler", amenable to extension and alteration from the very code it compiles. This has modularised the compiler into easily comprehensible sections, which are also easily debuggable sections. The `transform` declaration within a module, which introduces a function to be applied to any expression inside a module importing the one declaring the transformation, opens up the possibility of putting optimisations into modules; not only can these be standard generic optimisations, they can be optimisations that make use of deeper knowledge of the semantics of the functions exported from that module. A matrix manipulation module could contain optimisations that understand that multiplication by an identity matrix is redundant. I will explore these issues in more detail in the following sections.

3.3. Domain specific sublanguages

The ease with which arbitrarily complex new constructs can be added to Qube using the `meta` construct and its more user friendly but less flexible children such as the module system opens the floodgates to an entire generation of domain specific sublanguages.

Currently, domain specific languages are usually implemented in one of two ways:

1. *Writing the entire language from scratch.* This approach is usually taken in the belief that the language is so simple it will be easier to implement from scratch than to take any of the other options. The big problem is that, inevitably, feature creep sets in and the DSL ends up "reinventing the wheel" by finding its own unique way of implementing all the features of a fully fledge programming language, while still remaining crippled by its heritage. Perhaps Perl is a good example; it is now rarely, if ever, used as for the "extraction of reports"! UNIX scripting shells also come to mind.

2. *Skeletons wrapping a base language.* This is the approach taken by tools such as *lex* and *yacc* which are used in compiler generation. The underlying language is used to express the things it is best at, such as how to handle events, while the DSL is a skeleton with the base language snippets embedded in it. The DSL compiler works by converting the skeleton to the underlying language and then inserting the code snippets from the skeleton verbatim.

   This saves on the effort of reimplementing functions, modules, and other such standard constructs, yet is still relatively "clunky" and awkward to use. It should be fairly obvious by now that implementing said skeleton DSL as a Qube construct using `meta` processing would be a much more elegant and labour saving technique.

Let us consider some examples of how metaprogramming could be used to construct domain specific languages that are very tightly integrated with the base language, with little effort.

3.3.1. Parallel problem solving

Parallel programs can be expressed in many ways, perhaps the most useful of which is with "skeletons", which are very high level constructs expressing patterns of parallelism. Embedded within a skeleton will be snippets of sequential code responsible for handling individual subproblems, and are expected to be executed in parallel with each other on seperate processing nodes. The communication between these subprograms is defined by the type of skeleton used. For example, we might define `parallel-pipeline` skeleton that is used like so:

```
(public render
        (parallel-pipeline
                (fun (my-3D-model)
                        (transform my-3D-model viewpoint))
                clipWorld
                renderPerspectiveTransform))
```

The `parallel-pipeline` skeleton works much like a `fun` construct, with the constraint that it accepts exactly one parameter. The body of the construct is a list of functions. The semantic meaning of the parallel-pipeline is that the functions are composed and applied to each element of the input list, as if with the `map` function, and a list of the results collected to be returned; in this case, the function generated by the `parallel-pipeline` construct maps a list of 3D models to a list of bitmapped images.

Two implementations of `parallel-pipline` can be provided; one of which simply composes the functions given to it and maps the over the input list to yield the output, and the other of which actually exploits parallel processing. The former can be used in single processor environments, while the latter can be used (without needing to change the source code) in distributed multiprocessor environments.

Imagine we have access to a native library providing inter-processor message passing. We could run, on each "slave" processor, a program which accepts a message, executes it as a Qube program, then goes back to waiting for a new Qube program to execute. The program it receives and executes can, itself, use the message passing library.

With this runtime environment in place, the `parallel-pipeline` construct can rewrite to code that sends each function in the pipeline to a different processor. Each function will be wrapped in a loop which waits for a message, applies the function to the message, then passes the result to whichever processor is next in the pipeline; upon receipt of a special "stop" message on the input the loop exits. Having sent out these programs to all the slave processors, the code running on the master processor can then send the input list to the first processor in the pipeline, element by element, followed by a "stop" message, while at the same time collecting the results sent to it by the last processor in the pipeline and building them into a list which, when it has reached the length of the input list, can be returned as the output.

3.3.2. Finite state machines for control flow

Finite state machines are the most generic mechanism for expressing the flow of control in an imperative program. By defining a custom construct, it is possible to program directly with finite state machines. If defined carefully they can be very useful, especially in systems like communications protocol implementations, games, and user interfaces, where the concept of a state machine is very applicable to the problem domain.

There are many approaches we could take, but perhaps the simplest is to model a state machine as a list of named state, where each state is a function mapping the machine's parameter to a new parameter, and naming the next state to take control. The machine's parameter is a value of arbitrary type.

We might write our state machines like so:

```
(state-machine (Integer)
```

```
start (a (+ STATE 1))

a (if (> STATE 1)
                (a (- STATE 2))
                (terminate STATE))
)
```

We are forced to provide an initial state, called `start`, and the system automatically provides a pseudo-state `terminate` for us to pass control onto when we wish the machine to terminate and return the final value of its parameter.

It should be clear that this is little more than a slightly rearranged `let` construct inside a `fun` construct; it could rewrite to:

```
(fun (STATE :: Integer -> Integer)
        (let
                ((a (fun (STATE)
                        (if (> STATE 1)
                        (a (- STATE 2))
                        (terminate STATE))))
                (terminate (fun (STATE) STATE)))
        (a (+ STATE 1))))
```

However, the specially designed `state-machine` syntax is much clearer to follow.

3.4. Completing the evolution of Qube into a useful programming language

The minimal design of the basic Qube language, without `meta`, was chosen to save time on the implementation of fairly standard interpreter technology and to present a very simple, clean, canvas upon which to demonstrate the power of `meta`.

However, it is constructive to find the minimum set of features that would be required in the base language to allow the development of Qube into a fully useful programming language.

3.4.1. A compiler

Currently the result of preprocessing a Qube program, expressed in the base language, is interpreted. However, due to the triviality of the base language, it would be a simple task to write a compiler for any given host architecture. The most difficult implementation are would be the provision of a runtime garbage collector, since the existing Qube interpreter uses the underlying Scheme runtime system to handle memory management.

The base Qube language has no particularly special features, so existing compilation techniques such as conversion to continuation passing style [1] can be used. The conversion to CPS, itself, can be expressed as a source transformation and the base language reduced to an even more primitive form with continuations instead of functions, a task which Appel leaves to the first stages of his compiler, but which we can implement in standard libraries with `meta`!

3.4.2. Strict types

Interestingly, a strict type system requires no modification to the base language. If we extend our `fun`, `let`, and `module` constructs (the constructs we have so far defined that bind variables to values) to accept type annotations, we can define a processing phase that checks the correctness of these type annotations then strips them off; ideally, the `let` and `module` constructs would be implemented to pass these type annotations through to the `fun` constructs they use to implement themselves and then leave the type checker to only have to deal with `fun`. A suggested syntax for type annotations on `fun` declarations is:

```
(fun (x :: Int y :: String -> String) … )
```

This defines the function as accepting two arguments, `x` and `y`, with types `Int` and `String` respectively, with the whole function returning type `String`. This is acheived by giving the symbols `::` and `->` special meaning in `fun` construct argument lists.

Polymorphism in types can be expressed with a further extension to this, such as defining type names starting with `?` to be type variables with implicit scope of the function definition, or by introducing a specific `polymorph` construct to introduce new type variables and define a scope for them, possibly crossing several `fun` constructs.

We can also add type inferencing by adding a transform that looks for missing type declarations in `fun` constructs (the syntax has been chosen such that it can be unambiguously specified for some arguments and not others if necessary) and adds them with the standard unification algorithm. Again, since this is just part of a standard library, the user may choose to use a more powerful type system that is not always decidable.

Currently, the Qube base language is dynamically typed like Scheme. If the type checking phase (which would be easier to express in terms of the base language than higher up, anyway) were forcibly applied by the compiler, it would then have a proof that programs passed to it were correctly typed, and as such could then become completely untyped, like an assembly language.

3.4.3. Mutating operations and multithreading

The ability to perform mutating operations such as “I/O” or “updates” are very useful in a language, the former for communicating with the outside world beyond being given initial arguments and producing a result, and the latter for efficiently expressing certain algorithms.

However, it is not easy to provide such operations in a purely functional language. Making the base language impure would present many dangers; the partial evaluation system assumes that expressions have no side effects and so will merrily repeat them, execute them in arbitrary orders, execute them once and replace them with the result of that execution under the assumption that they will always reduce to the same value, and so on. Also, should expressions exhibit side effects, the design of source transformations based upon `meta` would be a much more complex affair. The consequences of reordering the execution of expressions would need to be carefully considered and expressed in the new construct’s documentation.

An extensive literature survey led to the conclusion that the best available technique for expressing mutating operations currently available, let alone with the restriction of remaining a purely functional language, is the use of *linear types* [2] to express mutable objects. The linear typing system used in that reference, as implemented in the language Clean [3], is awkward to

use due to syntactic constraints; it is somewhat too low level for normal use.

The fundamental idea of linear values, in summary, is that the language be extended with a "linearity" flag that can be set on bindings. A linear binding is much like a normal binding in that the variable is defined and given a value (or, in the case of a function argument, the assignment of a value is deferred until application). The difference is that the variable may only be referenced once in the flow of control within the scope of its definition; it may be referenced in both branches of an `if`, since those are mutually exclusive, but nowehere else. Also, if the variable is used as an argument in a function application, then the receiving argument of the applied function **must** be a linearly bound argument. Enforcing this requires a strict type checker. Whereas linear variables can be initialised from linear or nonlinear values, they can never be "converted back" to nonlinear variables.

Additionally, every variable has to be referred to **exactly** once. This means that any code using a linear variable will be forced to either return it as a value or pass it to a function that *can* destroy it. Likewise, due to the impossibility of referring to a linear variable twice, it cannot duplicate that value unless a duplication function for that type has been defined, or a function converting linear elements of that type into nonlinear elements. These two constraints together mean that linear values are exempt from garbage collection; they are *always* explicitly destroyed at some point by a `destroy` primitive.

Since any function can, if it declares a parameter as "linear", know that it has the one and only reference to a value, it could be so implemented as to destructively update that value without fear of spoiling the pure functionality of the language. Note that such an operation is still inherently impossible in a pure language, and has to be provided by the primitives for accessing each basic type which are defined as axioms of the language, along with the duplication and destruction operations mentioned earlier.

In a base language with linear types, the most flexible basic aggregrate data structure is the *array*. Arrays could be implemented with the usual set of primitives to create an array of a certain length, find the length of an array, and select the *n*th element of an array. Since we do not mind if users duplicate arrays, we provide a primitive to convert a linear array into a nonlinear array. To provide an efficient update operation, we create an `update!` primitive that accepts a *linear* array, an index, and a new value for the array element at that index, which returns a linear array result. This primitive is implemented using the efficient in-place updates functional languages are notorious for lacking, unlike the `update` function, which accepts a nonlinear array argument instead and returns a *copy* of the original array with that one element changed.

Very briefly, the advantages of linear values over other techniques such as monadic combinators start to manifest when we consider I/O and multithreading. I/O is implemented by passing the program a linear value of type World, which it must return upon completion (programs are functions of type *linear World*→*linear World*). No primitives are provided to create values of type World or to convert linear Worlds to nonlinear Worlds or to duplicate Worlds. Since the program has to return a World and cannot create new Worlds from anywhere, it must return either the World it was passed or the result of one of a suite of primitives that accept a linear World as their parameter and return a modified World. These primitives may include `print`, which accepts a World and a string, and returns a World in which that string has been output onto a display device, and `input`, which accepts a World and returns both a World in which enough time has elapsed for a human to input a string on the keyboard, and that inputted string.

The philosophy is clear; we treat the "outside world" as a value the program modifies to create a world in which the consequences of executing the program have unravelled. Treating interaction with the outside world the same way as we treat interactions with internal mutable state is an intuitively pleasing generalisation, if awkward to code with due to having to pass the World value everywhere.

Multithreading can be implemented in the World system by providing a function that executes multiple programs in parallel. This function itself must accept a linear World argument and finally return a linear World in which all of those programs have executed. For each thread, the World represents the entire outside world including all of the other threads; therefore, any inter-thread communication functions must be functions accepting a linear World and returning a new linear World in which that communication has occurred. It slowly becomes apparent that the passing of the World from function to function encapsulates the flow of time; every primitive function that touches the World has *before* and *after* defined for it by the dataflow of the World object. The way in which linear typing naturally expresses "causality" and "the flow of time" is, in my opinion, almost eerie.

Another benefit of using linear types for mutable values is that they enforce thread safety. Since a given linear value can never be duplicated, it may be controlled by only one thread of execution at any given time; using inter-thread communications control over the linear value may be passed over, but accesses to mutable variables are inherently serialised by the type system. If explicitly passing control over a mutable value between threads is too cumbersome, we can easily define a "shared state object" which enforces a kind of transaction structure; the shared state object is not itself linear, but it wraps a linearly typed value. A function is provided to obtain a nonlinear copy of the "state" at a given point in time - since it is specific to that point in time, it needs to accept and return a linear World. The shared state is considered to be a part of the "world", since it is outside of the current thread. In order to update the state, we can use a function that accepts a linear World, the shared state wrapper object, and a function of type *linearState*→*linearState*, which can be considered a *transaction* on the state. The special thread controlling the actual linear state value inside the shared state object accepts these transactions through some kind of inter-thread communication queue, executes them in arrival or priority order by passing them the linear state and then looping with the new value of the linear state replacing the old.

The downside of this is that an unweildy syntax is required. For example, a program that accepts an integer from the keyboard, adds that integer to a shared counter, and then prints out that value looks like this:

```
(fun (world :: linear World -> linear World)
        (let ((world1 (print!  world “Enter an integer”)))
        (let (((world2 number) (input-int!  world1)))
        (let ((world3 (modify!  world2 shared-counter
                (fun (x :: linear Int)
                        (make-linear (+ number (make-nonlin-
ear x)))))))
        (let (((world4 total-count) (get-state!  world3 shared-
counter)))
        (let ((world5 (print world4 (+ “Total count:  ” (int2string
total-count)))))
```

```
        world5))))))
```

We use nested `let` constructs to manage the passing of the value of the world from each operation to the next, then finally return the resulting world.

This is where customisable program transformation proves its worth! We can define an improved notation for "procedures", a name given to functions that operate upon mutable state and traditionally bound to names suffixed with an exclamation point, which handles the nitty gritty of passing linear values around:

```
(proc ( World | )
        do (print!  "Enter an integer")
        set number = (input-int!)
        do (modify!  shared-counter
                (fun (x :: linear Int)
                        (make-linear (+ number (make-nonlin-
ear x)))))
        set total-count = (get-state!  shared-counter)
        do (print!  (+ "Total count:  " (int2string total-
count)))
        return)
```

The intent there is much clearer; the only item that needs explanation is that the syntax for a `proc` argument list is an extended version of the typed `fun` construct introduced above, which specifies a linear type as the first element of the argument list followed by a `|` seperator (purely for clarity), then any nonlinear arguments the procedure requires. If the procedure specifies a return type, it will actually return a list with the linear output value first and the returned value of the procedure second.

The body of the `proc` is a list of statements, each starting with a symbolic name. `do` executes a procedure that returns nothing beyond it's linear state variable, `set` executes a procedure that returns additional values and binds those returned values to names, and `return` formally marks the end of the procedure, returning the linear state variable and any other return values specified.

A more complex example, too unweildy to express in plain nested `let` notation, is:

```
(proc ( World | name :: String -> String)
        do (print!  (+ Is your name  name ?))
        set result = (input!)
        if (= result Y) then
                return name
        else
                do (print!  What is your name, then?)
                set newname = (input-with-default!  name)
                return newname
        endif
)
```

Here arguments to procedures are demonstrated, along with an `if` statement.

Ideally, this would be implemented in such a way as to allow the easy addition of new constructs; perhaps, using a `meta` construct, the `proc` construct compiler function could handle a statement of type `XXX` by finding and invoking a function called *`XXX`*`-statement`; this is trivially expressed as:

```
((meta (string2symbol (+ statement-type -statement))) statement)
```

A construct for easily dealing with multiple linear states in a proc is slightly tricker to define, but not awesomely so; however, it is certainly beyond the scope of this report.

3.4.4. Seperate compilation

As things stand, the Qube module system works by including the full body of every referenced module using `extern` constructs. Although this means that optimisations such as inlining are applied over the entire program, it is generally inefficient since it means that compilation will be a slow process and the resulting machine code program quite large. It would be much nicer if we supported a form of seperate compilation, where modules could be compiled independently and then linked at run time, allowing for operating system support of shared libraries to reduce the memory footprint of Qube programs considerably.

Luckily, this is made easy by relatively minor changes in the Qube compiler and runtime primitive library.

Currently, the Qube compiler accepts top level module definitions. A slightly altered version of it can be created that accepts normal module defintions and compiles them to a Unix shared library (.so) or Windows dynamic library (.dll) file, and a "stub" Qube module file that exports the interface of the source module, but implements it by using the operating system's primitives to load the shared library file and obtain pointers to the public bindings from the shared library. This will require extensive support from the run time library, especially with regards to enforcing type safety by checking the embedded type signature in the shared library in case it has been swapped for another, but will not be at all difficult. All of the generation of stub code can be produced by `meta` powered rewriting.

4. Conclusions and Future Directions

Hopefully, the reader will now have an idea of how a very simple addition to a pure functional programming language can completely reshape the way the language works, adding dimensions of expressive power hitherto barely imagined.

Needless to say, research time needs to be applied to the implementation of the base language extensions described above, as well as to other issues such as the implementation of an object oriented type system. It is hoped that a descendent of Qube will become an invaluable tool in the prototyping of new programming language paradigms and constructs in the years to come.

Other works in the area of program transformation tend to focus on very specific uses of transformations, such as conversion into continuation passing style or eliminating multiple-

argument functions by using currying, rather than providing a practical modular framework for expressing them. They also tend not to consider how program transformations can be used to express language constructs in terms of a minimal base language, or the quite astounding implications of this approach for language design.

Continued research into the `meta` construct will be carried on as part of my ARGON project, which I have been working on for more than five years now. The eventual goal of ARGON is to develop a so-called "next generation computing system"; the programming language part of this is still under theoretical development, as I am focussing on communications issues at the time of writing. Interested parties are welcome to contact the development mailing list, `argon@argon.cx`.

References

[1] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press.

[2] Henry Baker. Linear Lisp. URL ftp://ftp.netcom.com/pub/hb/hbaker/LinearLisp.html.

[3] The Concurrent Clean home page, Software Technology Research Group, University of Nijmegen. URL http://www.cs.kun.nl/~clean/.

[4] Donald Knuth. *Literate Programming.* Center for the Study of Language and Information.

[5] Revised^5 Report on the Algorithmic Language Scheme. URL http://www.cs.rice.edu/CS/PLT/packages/doc/r5rs/index.htm.

[6] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press.