Myth Technical Report

Greg Rowe

Abstract

Myth is a programming language that is an extension of C. Myth adds modules, interfaces, tuple returns, and bit sets. These features are very helpful in nearly any programming environment. The Myth language as a whole is a good choice for embedded systems where resources are limited. Modules help organize code into logical groupings. Interfaces provide an automated mechanism to write reusable code. Tuple returns provide a simple mechanism for returning multiple values from a function. Bitsets make it easy to define logical groups of bit flags. Bisets are particularly useful in code that interfaces with hardware.

The advantage of Myth is modules, tuple returns, and interfaces without the overhead of full-blown object orientation. Myth has been implemented as a preprocessor that produces C source code.

1. Background

C is a very popular programming language. It is a small language with many efficient implementations. C is also highly portable. The syntax and semantics of C have influenced the design of many different programming languages. Some languages, like Myth, add features to C while leaving the base implementation unchanged.

Objective-C is a language that adds Smalltalk-like object oriented features to C. The first implementation was by Brad Cox in the early 1980s [fedor]. Objective-C is a strict superset of C. A programmer could compile unmodified C code directly with an Objective-C compiler. It is a compiled language that works well with existing libraries written for C. C++ adds many features to C with object oriented support being the most significant addition. C++ was created by Bjarne Stroustrup.

Java is a relatively new language that is very popular. Java shares a lot of syntax with C and C++. Other than syntax Java shares little with C. Java is intended to be run under an interpreter using byte code. This is a significant difference between C, Objective-C, and C++.

Limbo is another relatively new language that borrows from the C syntax. Like Java the current implementation is an interpreter that operates on intermediate code. "Limbo is a programming language intended for applications running distributed systems on small computers." [ritchie]

S-Lang is another language that adopts a C-like syntax. S-Lang is implemented as an interpreter that can easily be embedded into an existing application. It was originally developed as an extension language for the Joe text editor but is now embedded in many other applications. [davis]

Modular programming is almost universally encouraged as a good programming technique. Most languages have features to assist programmers with organizing their code into modules. Some versions Pascal for example have `Module` as a keyword. Java has a notion of packages which can be used to organize groups of related code into a common bundle.

Object oriented programming has made the idea of interfaces popular. Even programming languages like Limbo that are not considered to be object oriented have adopted the idea. Interfaces allow the programmer to define a set of actions or methods. Other parts of code are said to implement an interface. Code can then be written against an interface. Code written against an interface can operate on any other code that has implemented that interface.

No programming language is perfect for every task. C does a good job under a wide range of applications but could benefit from a few extra language features. Myth is certainly not the first language with the intent of bolting on a few additions to C. Objective-C exists to add object oriented mechanisms to C [fedor]. C++ also builds upon C but isn't strictly a super set of C [fedor].

Myth builds upon C because it is a powerful and simple language. C is used for many types of programming tasks ranging from operating systems and device drivers up to large scale applications. There are also many libraries available to C programmers.

Myth is a strict super set of C. It adds a few language features found in other languages in an effort to make some common tasks in C easier to accomplish and to promote easily readable and maintainable source code.

Table 1. Language Comparison

Language	Code Grouping/Modules	Interfaces	Tuples	Object Oriented	Interpreted
Myth	Yes	Yes	Yes	No	No
C	No	No	No	No	No
Java	Yes	Yes	No	Yes	Yes
C++	No	Yes	No	Yes	No
Objective C	No	Yes	No	Yes	No
S-Lang	Yes	No	Yes	No	Yes
Limbo	Yes	Yes	Yes	No	Yes

2. The Myth Programming Language

The features Myth adds to C are bitsets, tuple returns, modules, and interfaces. Bitsets provide a convenient tool for defining and organizing bit masks. Tuple returns provide a logical and consistent method for returning multiple values from a function. Modules are a tool for organizing source code that all operates on a common set of data. Interfaces make it easier for the programmer to write reusable code.

2.1. Bitsets

Myth bitsets provide a simple and convenient mechanism for grouping related bit masks. The syntax is similar to that of a C `enum`.

2.1.1. Bitset Rationale

Bitsets are present in Myth because they provide a convenient and useful mechanism for managing bit masks. A lot of low level code uses very long header files full of preprocessor definitions for bit masks. Managing these flags is often cumbersome because they are not linked together in any way.

Bitsets in Myth are a misnomer. Bitsets are really sets of bit masks, not sets of bits.

Sometimes a programmer is faced with an environment that has a very small amount of memory. In this sort of situation a smart programmer will pack multiple boolean values into a single entity. Preprocessor defines, enums, and structs with bit fields can all be used to pack booleans into smaller spaces.

The technique of packing multiple boolean values into a `struct` with bit fields of one bit has many drawbacks. Few real machines allow access to memory locations smaller than 8 bits in size. Therefore it is required that the compiler generate the code necessary to extract the particular bit that is referenced in the struct instance. The amount of code that is generated for these operations can be surprising. It is likely that the amount of executable memory used can increase using this technique.

A more serious drawback of this technique is that compilers often create structs with padding bytes for performance reasons. Take for example the following `struct`:

```
struct packed{
        unsigned int p1 :1;
        unsigned int p2 :1;
        unsigned int p3 :1;
        unsigned int p4 :1;
        unsigned int p5 :1;
        unsigned int p6 :1;
        unsigned int p7 :1;
        unsigned int p8 :1;

};

struct packed packedBooleans;

printf("%d\n", sizeof packedBooleans);
```

In the example above, when compiled on different architectures, will have a different size of `packedBooleans`. On the author's machine the size happens to be 4 bytes. A naive programmer may think that the size of the `struct` should be 1 byte but that is not the case under most conditions.

Another technique is to use the preprocessor or `enum` declarations to define bit masks to mask out individual bits in a packed value. These techniques are portable and do not suffer from architecture specific padding issues. These techniques also make the amount of code necessary for extracting individual items very clear as the programmer must write the code. These techniques however require a lot of manual work by the programmer and are not easy to maintain.

Neither the preprocessor nor the `enum` mechanism are convenient from a maintenance perspective. Items declared in an `enum` are automatically incremented but for bit sized masks this is not desirable. Each item in the `enum` must be overridden by explicitly specifying a value. When using the preprocessor all values must also be explicitly defined and there is no syntactical grouping like the enum method provides. If the programmer needs to rearrange the bitwise locations of the flags they must modify all values in the set. This is both inconvenient and error prone.

The Myth bitset provides a simple mechanism for declaring sets of bit masks. Extracting and setting values for a packed boolean value using a Myth bitset is very similar to using the `enum` or preprocessor technique. Where the Myth bitset is superior is that it provides a common naming convention that groups similar masks with a common name as a prefix. More importantly, in the case of a packed boolean, each item in the bitset increases in a bitwise fashion. The first item in a Myth bitset has only the first bit set. The second item has the second bit set. This pattern continues for the entire bitset. If at any time the position of the value must change only the textual position of the members must change. The values of the members are generated by Myth.

A set of bit masks often can be for masking out different parts of a hardware register. These types of flags are almost always defined using hexadecimal notation. For all but the most experienced programmers converting hexadecimal to binary is not something that can be done easily in their head. Hardware documentation usually describes register contents using decimal bit ranges, not hexadecimal numbers. This translation between different notations is error prone. Locating an error of this type is a very difficult task because the code does not resemble the hardware documentation. Specifying ranges of set bits using decimal values can make the code more readable by more closely resembling hardware manuals.

Myth bitsets provide the programmer with a tool for logically grouping related bit masks and a convenient mechanism for specifying what bits in a particular mask are set. The ability to specify a range of set bits in a mask can make the code more closely resemble hardware documentation. Myth bitsets can be used to deal with packed boolean values in a portable and easily maintainable manner.

2.1.2. Bitset Grammar

A Myth bitset has syntax similar to that of a C `struct`. A bitset begins with the keyword `bitset` followed by an identifier. The body of a bitset is surrounded by curly braces. The body consists of a list of identifiers each with an optional bit range.

A bit range is a comma separated list of ranges. A ranges is an integer followed by '..' followed by another integer. The integers are interpreted as bit positions such that every bit position in the range is set to 1. Any bit not specified in the range is set to 0 for the mask.

```
bitset_decl
: 'bitset' IDENTIFIER '{' bitset_member_LIST '}'


bitset_member
: IDENTIFIER ';'
| IDENTIFIER ':' range_item_CLIST ';'

range_item
: CONSTANT RANGE_SEPERATOR CONSTANT
```

2.1.3. Bitset Example

The following example illustrates a definition that would be useful for extracting data from a packed boolean value such as a 32 bit integer where each bit represents a single boolean value. In the packedBoolean example three bit masks are created. The first mask, bitZero, has only the first bit set. The second mask, bitOne, has only the second bit set. This pattern continues for the remaining items in the bitset. These masks can be accessed using the bitset name as a prefix and the mask name as a suffix. In the example `justBitOne` will have the value of 0x1 assigned to it. The mask bitTwo would have a value of 0x4.

```
bitset packedBoolean
{
        bitZero;
        bitOne;
        bitTwo;
}

unsigned int exampleInt = 0xff;
unsigned int justBitZero = exampleInt & packedBoolean_bitZero;
```

This next example illustrates using a bitset for accessing a memory mapped register for some hardware device. The range feature for bitset members is utilized to define what bits are set in the masks that are created. In the example the masks are used to extract individual pieces of information from a single memory mapped register.

```
bitset registerA
{
        validResult: 0..1;
        resultValue: 2..31;
}

unsigned int *regA = 0x08000000;
int result = 0;

if( *regA & registerA_validResult){
        result = *regA & registerA_resultValue;
}
```

The final example illustrates all features of a bitset. In a bitset one can mix masks that have ranges with masks that automatically increment the set bit location (like in the first example for a `packedBoolean`). It also illustrates that it is legal to specify multiple set bit ranges on a single mask. In this example the output of printf would be:

```
bitZero=1  complexMask=1e1c00 bitOne=2 bitTwo=4
```

```
bitset complexBs
{
        bitZero;

        complexMask: 10..12, 20..17;

        bitOne;
        bitTwo;

}

printf("bitZero=%x  complexMask=%x bitOne=%x bitTwo=%d\n",
        complexBs_bitZero,
        complexBs_complexMask,
        complexBs_bitOne,
        complexBs_bitTwo);
```

2.2. Tuple Returns

Tuple returns are a language feature borrowed from languages like Limbo and S-Lang. Tuple returns provide a logical syntax for returning multiple values from a function.

2.2.1. Tuple Return Rationale

C functions are allowed to return a single value to a caller. If a function must return multiple values there are a few mechanisms that a programmer can utilize to achieve that goal. This sort of situation arises frequently.

Many standard C function calls return extended error information via the global errno variable. The `fopen` call for example signals a failure to its caller by returning a `NULL` `FILE` pointer. To determine why the call failed the global `errno` variable must be examined. This technique is riddled with potential problems. First, it is not reentrant. When programming in an environment with shared memory, such as using multiple threads, the global value can't be relied upon. In addition it is not clear at a glance that `errno` has anything to do with the `fopen` call. A novice programmer who may not have read their operating system programming manual carefully would not think to consult the `errno` variable for extended information.

One technique for providing multiple return values is to use an out parameter. An out parameter is a parameter passed into a function that is a pointer. This makes for a reentrant solution but results in code that can be difficult to read. The code becomes less logical because a parameter that is passed into the function essentially becomes a value passed out of the function. It would be clearer if all function parameters were input, and the return value was the only output.

Using a `struct` C programmers can return any number of values from a function. This technique is very expensive in terms of programmer time. The programmer must define the structure and make that definition available to any client that wishes to call that function. These steps can be worth the programmer's time if there are a sufficiently large number of values that need to be returned. However, when there are a small number of return values, such as a failure notice and a reason code, it isn't worth the effort of defining a struct. In these cases lazy programmers will use poor techniques such as global values or possibly dropping error information completely.

The following is C code to wrap the `fopen` call to return a `FILE` pointer as well as an error code.

```
/* In a header file named myfile.h */
#include <stdio.h>
struct fileReturn_t {
        FILE *file;
        int  error;
};
struct fileReturn_t fileOpen(const char *path, const char *mode);
```

```
/* In a source file */
#include "myfile.h"
#include <stdio.h>
#include <errno.h>

struct fileReturn_t fileOpen(const char *path, const char *mode)
{
        struct fileReturn_t fileInfo = {NULL, 0};

        fileInfo.file = fopen(path, mode);
        if(fileInfo.file == NULL){
                fileInfo.error = errno;
        }

        return fileInfo;
}

/* In a second source file that uses the fileOpen call */
#include "myfile.h"

int main(int argc, char *argv[])
{
        struct fileReturn_t fileInfo = {NULL, 0};
        fileInfo = fileOpen("/etc/shadow", "r");
        if(fileInfo.file == NULL){
                fprintf(stderr, "%s\n", strerror(file.error));
        }

        /* rest of code ommitted */
        return 0;
}
```

Tuple returns provide a mechanism to solve this problem. Using tuples a function can define a list of return values in nearly the same manner that the input parameters are defined. This allows the programmer to make input values and output values very clear to the caller. This mechanism is reentrant and costs the programmer no more effort than declaring an extra return value. Tuple returns are not appropriate in situations where a large number of values must be returned. In that sort of situation a `struct` is more appropriate as the programmer can name each item in the struct. In addition a `struct` is a true data type and can be passed between functions as a single parameter.

Tuples in Myth are not data types. A tuple can only be used to return multiple values from a function. This was a deliberate decision to prevent potential abuse of tuples. In most cases when a tuple would be desired as a data type it would be more appropriate to declare a `struct`.

Tuple data types and returning multiple values from functions is not new. S-Lang allows for an arbitrary number of return items from a function [davis]. Limbo has a tuple data type which allows for simple returns of multiple values [ritchie].

2.2.2. Tuple Return Grammar

A function that returns a tuple is declared much the same way that an ordinary C function is declared. Instead of providing a C type for the return value of the function the keyword `tuple` is specified. Following the `tuple` keyword are a pair of parenthesis. Inside the parenthesis a parameter list is declared. The parameter list is a comma separated list list of types.

```
tuple_type
: ’tuple’ ’(’ parameter_type_l ’)’
```

2.2.3. Tuple Return Example

The example below illustrates a rewrite of the previous example of opening a file using a tuple return. tuples in Myth.

```
/* In a header file named myfile.h */
#include <stdio.h>

tuple(FILE*, int) fileOpen(const char *path, const char *mode);


/* In a source file */
#include "myfile.h"
#include <stdio.h>
#include <errno.h>

tuple(FILE*, int) fileOpen(const char *path, const char *mode)
{
        int error = 0;
        FILE *file = NULL;
        struct fileReturn_t fileInfo = {NULL, 0};

        file = fopen(path, mode);
        if(file == NULL){
                error = errno;
        }

        return tuple(file, error);
}

/* In a second source file that uses the fileOpen call */
#include "myfile.h"

int main(int argc, char *argv[])
{
        FILE *file = NULL;
        int error = NULL;


        tuple(file, error) = fileOpen("/etc/shadow", "r");
        if(file == NULL){
                fprintf(stderr, "%s\n", error);
        }

        /* rest of code ommitted */
        return 0;
}
```

2.3. Modules

Myth modules provide a mechanism for grouping similar code that is not present in C.

2.3.1. Modules Rationale

Writing a computer program involves taking a problem and breaking it into smaller more easily solvable pieces. What separates great programmers from the others is their ability to break down a problem in a simple and organized manner.

Modular code has long been stressed as good programming technique. Brian Kernighan once wrote "controlling complexity is the essence of computer programming" [kernighan-plauger]. Eric Raymond recommends making programs "out of simple parts connected by well defined interfaces" so one can have "some hope of upgrading a part with out breaking the whole" [raymond]. A program module should contain code and data that are all related in some fashion. Many languages have explicit mechanisms for organizing code in this sort of manner. Pascal for example has a package mechanism. Another example is Java which provides packages and classes in a sort of two tiered approach. Limbo is a very module oriented language [ritchie]. C on the other hand doesn’t have such an obvious mechanism. Instead to achieve the same sort of effect a programmer organizes code into separate compilation units. A compilation unit is a file.

A logical result of organizing related code into files is to prefix function and variable names with the name of the file to prevent symbol clashes and to identify the items as belonging to a particular compilation unit. It can also be desirable to define a common data type that each function operates on. This is similar to object oriented programming because each function operates on a particular object. This differs from object oriented programming in that there is no mechanism for inheritance. C does not have mechanisms for data protection like most object oriented languages do. The best C has to offer is the `static` declaration specifier. This specifier tells the linker that the symbol is not accessible outside of that compilation unit.

When using this type of coding style the common data to each function very often ends up being a `struct`. It is logical that the name of that `struct` references the name of the file. The name of the file is also the prefix name for all or most of the functions. When organizing code like this it can become tedious to repeatedly type the name of the module. The programmer must essentially type the name of the module at least twice when declaring a new function. It must be entered once for the function name prefix and a second time as a parameter to the function.

C++ provides a namespace feature that allows for a flexible naming prefix mechanism. Myth takes a more simplistic approach with its module mechanism. A Myth module removes the tediousness of typing and retyping the name of the module over and over. Any function defined in a module automatically has the name of the module prepended to it. In addition every module has a `struct` associated with it. A pointer to a `struct` of the module type is passed to every function declared in that module. Module functions have access to module state data via a `self` pointer. This is not unlike the `this`pointer in C++. Unlike most object oriented languages Myth offers no method of protection for modules variables in the `self` section nor for the functions in a module. All Myth module state and functions are available to module clients. This is similar to an `adt` (abstract data type) in Limbo [ritchie].

2.3.2. Modules Grammar

Modules are declared and defined prior to use. A module declaration consists of the keyword `module` followed by an identifier. Following the identifier is an optional declaration of interfaces that the module implements.

The body of a module declaration consists of a `self` declaration and function declarations. The `self` section can be at the beginning or the end of the body.

The `self` section is a surrounded by a set of curly braces. Inside the body anything that can be declared in a C `struct` may be declared. All modules must have a `self` section but it may be empty.

Modules may have a storage class specifier. When a storage class specifier is present it is applied to all functions contained in the module.

The definition of a module is identical to the declaration of the module with a few exceptions. There is no `self` section inside a module definition. Functions in the module declaration have bodies. In all other respects the syntax is the same as a module declaration.

```
module_decl
: storage_class_specifiers_OPT basic_module_decl

basic_module_decl
: MODULE IDENTIFIER  module_decl_body
| MODULE IDENTIFIER IMPLEMENTS id_or_typename_l module_decl_body

module_decl_body
```

```
: '{' self_decl module_function_decls_OPT '}'
| '{'  module_function_decls self_decl '}'


self_decl
| '{' struct_declaration_l_OPT '}'
```

2.3.3. Module Example

A module is declared and then defined prior to use of the module from client code. The module declaration typically resides in a header file while the definition resides in a source file.

The examples below show different ways that a module can be declared. `exampleModule1` and `exampleModule2` are equivalent. The `self` declaration can appear at the beginning or the end of a module declaration. The third module declaration will generate a syntax error from the Myth preprocessor. The syntax error is because functions are declared above and below the self declaration. The `self` section must appear at the beginning or the end of a module declaration. This is required because it makes a large module declaration easier to read. A programmer can locate the `self` declaration in two steps at most. It is also not valid to declare multiple selfs for a module. Anything that can be declared in a `struct` can be declared in the `self` section of a module declaration.

Although it is not illustrated in the example storage class specifiers are allowed on a module declaration. The `static` specifier for example can be used to make all of the member functions `static`.

When declaring and defining modules the `implements` clause can be used to declare what interfaces a module implements. An `implements` clause is optional and only interfaces that have been defined are allowed to be listed. For a module the `implements` clause determines the generation of initialization code for interfaces. This is discussed further in the Interfaces section.

```
module exampleModule1
{
        /* self */
        {
                int state1;
                int state2;
        }

        void printSelf();
}

module exampleModule2
{
        void printSelf();
        /* self */
        {
                int state1;
                int state2;
        }
}


module invalidModule
{
        void printfSelf();

        /* self */
        {
                int state1;
                int state2;
        }

        void anotherFunction();
}



module modWithInterfaces implements interface1, interface2
{
        {int state;}
}
```

In this example `modWithInterfaces` is declared. It uses the optional `implements` keyword. The `implements` keyword declares what interface or interfaces a particular module implements. If the interfaces in the `implements` list have not yet been declared the Myth preprocessor will generate an error. This error is generated because the preprocessor will automatically declare all the functions that are declared for the interfaces. In addition the preprocessor will also generate one initializer function for each interface.

Since `modWithInterfaces` implements two different interfaces two different interface initializer functions will be created. This allows two different interface instances to be initialized against the same module instance.

```
module modWIthInterfaces implements interface1, interface2
{
        void interface1Function()
        {
                printf("interface1Function, my state=%d\n", self->state);
        }

        void interface2Function()
        {
                printf("interface2Function, my state=%d\n", self->state);
        }
}
```

This example shows a module definition. It is possible to define functions in a module that are not declared in the module. In the case of a module definition the `implements` list causes the code for the interface initializers to be generated. If any interface in the `implements` list does not exist the Myth preprocessor will generate an error. As with the module declaration the module definition can have a storage class specifier. As an example the `static` storage class specifier can be used and will make all functions in the module `static`.

2.4. Interfaces

Interfaces are a feature borrowed from other languages. In Myth interfaces allow for different types to be treated as a common type with known capabilities.

2.4.1. Interfaces Rationale

The concept of interfaces is not new to Myth. Java, C++, Objective-C, and numerous other languages provide this features similar to the Myth interface. In Myth an interface is a set of function pointers and an associated state. This differs from languages like Java and Objective-C whose interfaces have no state. The state of a Myth interface is always a pointer to an instance of a module so it could be argued that Myth interfaces do not have any state of their own.

An interface on its own has little meaning in Myth. To use an interface it is expected that the programmer will define a module that implements that interface. By defining a module that implements a specific interface the Myth preprocessor will generate code that will initialize an interface instance. After the interface has been initialized interface calls can be made. When an interface is initialized Myth assigns the function pointers to the corresponding functions defined for that module. The `self` pointer of the interface is also assigned to a pointer to a module instance. This allows an interface call to have access to the module state by way of the `self` pointer.

When an interface call is made the Myth preprocessor automatically passes the state or `self` pointer to the function being called. In addition each module function automatically casts this pointer to the correct type. Myth syntax requires that interface calls be tagged with an `@` character. This alerts a source code reader to the fact that a function call is being made indirectly.

By writing code that operates on interfaces it is easier to write reusable code. For example a sorting algorithm needs only to know how to compare individual items. A sorting algorithm could be written to operate on a `comparable` interface. The comparable interface would provide a function that compares two items.

2.4.2. Interfaces Grammar

An interface declaration consists of the keyword `interface` followed by an identifier. The body of an interface is surrounded by curly braces. In the body of the interface can be any number of function declarations.

```
interface_decl
: INTERFACE IDENTIFIER ’{’ module_function_decl_SLIST ’}’
```

2.4.3. Interface Example

```
interface example
{
        char* toString();
}
```

The example above declares an interface named `example` that contains a single function. This declares a new variable type of the name `example` to be available to the program. It is not legal to declare data in an interface. Only function declarations are allowed. Using a variable of type `example` would require initialization of the interface with an initializer function from a module that implements the `example` interface. The next example illustrates how to use an interface.

```
module exampleModule implements example
{
        char* toString(){
                return strdup("This is exampleModule reporting for duty");
        }
}

int main(int argc, char *argv[]){
        char *output = NULL;
        example exampleInterfaceInstance;
        exampleModule exampleModuleInstance;

        exampleModule_example(&exampleModuleInstance, &exampleInterfaceInstance);

        output = @exampleInterfaceInstance.toString();

        printf("%s\n", output);
        free(output);
        return 0;
}
```

In this example an interface is used in the function `main`. Before an interface is used it must be initialized. The Myth preprocessor generates the code necessary to initialize an interface. The name of the initializer function is the name of the module followed by an underscore and finished with the name of the interface. An interface initializer function takes two parameters. The first parameter is a pointer to an existing module instance. The second parameter is a pointer to an interface instance. Inside the initializer function the correct function pointers are assigned to the interface instance as well as the `self` pointer. The `self` pointer is a pointer to module instance that is passed in. The function pointers refer to function defined in the module.

Once the interface instance is initialized it can be used. Myth requires that interface calls be tagged with the `@` character. This tag notifies the Myth preprocessor that the function call is actually an interface call. The preprocessor can then ensure that the self pointer is passed into the function that is being called.

2.5. Features Not Included

The choice of features that are not included in Myth is nearly as important as the features that were chosen to be put into Myth. Many features that are found in other languages are purposely absent from Myth. Among them inheritance, and function overloading are particularly noteworthy for their absence.

Myth modules and interfaces provide a limited amount of object oriented functionality. Most languages that have object oriented functionality provide language features for inheritance. Inheritance is intentionally absent from Myth. The primary reason for a lack of inheritance is because it is difficult for programmers to use inheritance properly. Abuse of inheritance leads to code that is difficult to maintain [asynder]

Exceptions are also not present in Myth. One goal of Myth is to remain small, like C is. Myth makes certain programming techniques easier for the programmer but does so without any further requirements on the runtime environment.

3. Design of Mythpp, the Myth Preprocessor

The Myth pre-processor was implemented originally using C and later ported to Myth. Many tools were used to develop `mythpp`.

3.1. C as the Output Language

Using an output language such as Objective-C or C++ would have made the implementation for interfaces and modules much easier. C was chosen as the output language for the Myth preprocessor for a variety of reasons. There are C compilers for nearly every architecture and operating system. The GNU Compiler Collection version 4.0 reaches a more than 30 architectures [gcc]. Another reason is that Myth tries to be small and efficient. If a language like C++ or Objective-C were used as the output language then Myth programs would require their runtime systems which tend to be larger than for C. Finally, one goal for this project was to make it self hosting. It would be far more difficult to make a self hosting Myth preprocessor if the output language was anything other than C.

3.2. Preprocessor vs. Compiler

Since Myth uses C as its base it seemed logical to implement Myth as a preprocessor. Being a preprocessor provides many advantages. Typical Myth programs are expected to have a greater amount of pure C code than code that uses Myth constructs. Therefore it is necessary only to translate Myth specific constructs into their C equivalent code. Bjarne Stroustrup's first C++ compiler, cfront, was implemented as a preprocessor as well [stroustrup].

This technique has a significant advantage because it can leverage all of the semantic analysis of an existing C compiler. Therefore the complexity of the Myth preprocessor is greatly reduced as the only semantic analysis that is required is for the limited number of Myth constructs.

One drawback to this approach is that the error messages from the C compiler may be slightly cryptic as they refer to the C syntax and not the Myth syntax. The `#line` directive allows for the C compiler to reference the correct file and line where a semantic error has occurred but meaning can be lost in the translation.

With correct usage of the `#line` directive it is possible to debug Myth code directly. Using existing tools such as gdb it is possible to set breakpoints directly from the Myth source files. When stepping through source code gdb will display the lines from the original Myth source code instead of the code that was actually compiled.

3.3. Tools Used

A few tools play an integral role in mythpp. Without these tools it is likely that mythpp would not have been completed.

3.3.1. Public Domain C Grammar

Jeff Lee posted a Bison compatible ANSI C grammar and partial lexical specification that conformed to the ANSI C draft from April 30th, 1985. This grammar was chosen as the basis for the Myth preprocessor for a variety of reasons [lee].

It is not a trivial task to write a parser specification for the C programming language. Under the given time and complexity constraints it would not have been possible to complete the Myth preprocessor if a parser specification was written from scratch. One side effect of choosing this grammar was that the version of C being recognized is closer to C89 than C99. Originally the goal of Myth was to extend C99.

3.3.2. Flex and Bison

The decision to use GNU flex and GNU bison primarily arose from the choice of using Jeff Lee's C grammar. Since his parser specification and lexical specification are compatible with GNU bison and GNU flex it was natural to chose to use these tools to build the Myth preprocessor. It was also convenient that these tools use C. It made it possible to use Myth constructs directly in the parser specification during the porting effort.

3.3.3. Boehm-Demers-Weiser Garbage Collector

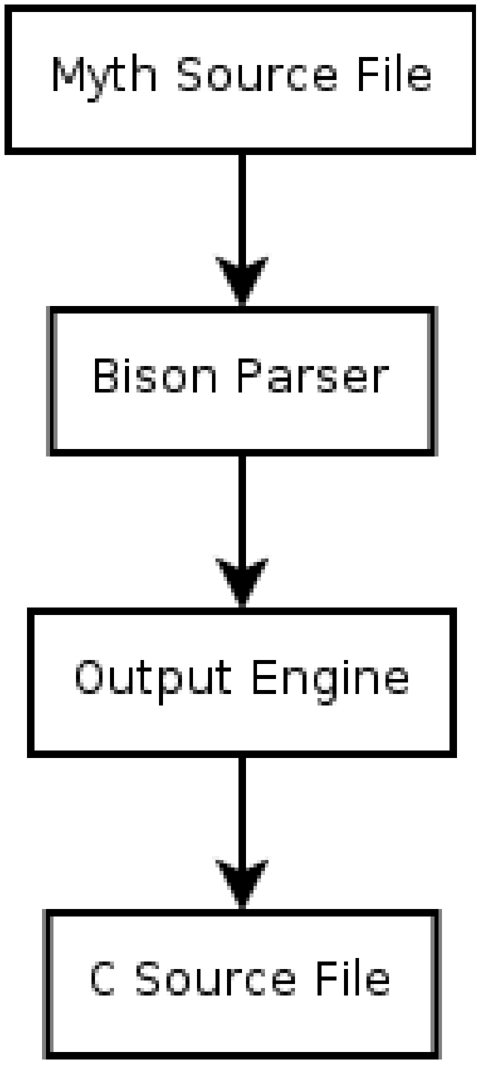
Nearly any non-trivial C program destined to run on a personal computer (as opposed to an embedded system) can benefit from a garbage collector. The Boehm garbage collector is a C library that can be used with C and C++. This library frees the programmer from worrying about cleaning up dynamically allocated memory. The Myth preprocessor has few locations where it actually deallocates memory and thus few possible places where a leak can occur. However, the use of the garbage collector makes dealing with local string construction easier. Since the Boehm garbage collector is trivially easy to use there was little reason not to use it [boehm-demers-weiser].

3.3.4. Glib Utility Library

The Glib utility library is a C library originally written for the GTK+ widget toolkit [glib]. Glib provides many features that are useful to a C programmer. Although its roots lie in a graphical widget toolset the library is strictly utility code with no dependencies on graphical environments. The Myth preprocessor primarily uses glib as a source of data structures. Myth uses the glib string types, linked lists, hash tables, and pointer arrays extensively. The pointer array is the most heavily used glib data structure in mythpp.

3.4. Program Flow

From a high level perspective the Myth preprocessor is rather simple. The input is in the form of a file. Once the command line options are parsed the Bison generated parser runs on the input file. Assuming there are no syntax errors the parsed data is then passed to the output code generator. The output code generator iterates over a list of items that need to be translated into C. For all cases except function bodies the code generator directly calls the functions that generate the C output. In the case of a function body it has its own engine that has a list of items that need translation or direct output. Since the Myth extensions are so simple and there is very little semantic analysis required the semantic analysis is done during code generation.



3.5. Generating Output

mythpp is centered around the idea of an output engine. The output engine generates C code from the Myth code that was extracted from the source code. The output engine also echoes back C source where no Myth extensions are used.

3.5.1. Outer Level Output Engine

Output is generated in two major loops. The outer loop iterates over a list of parsed Myth data. The parsed data has an associated location where it was defined in the source code file. This main loop iterates over these locations. The locations are only the locations where Myth syntax was parsed. For any location that requires translation to C line directives are used to refer back to the original source locations. For the gaps where pure C code was parsed this engine will echo back the locations from the source file. These sorts of locations are determined by examining the gaps between the locations of Myth parsed data. A memory mapped file makes echoing sections of the source file very easy.

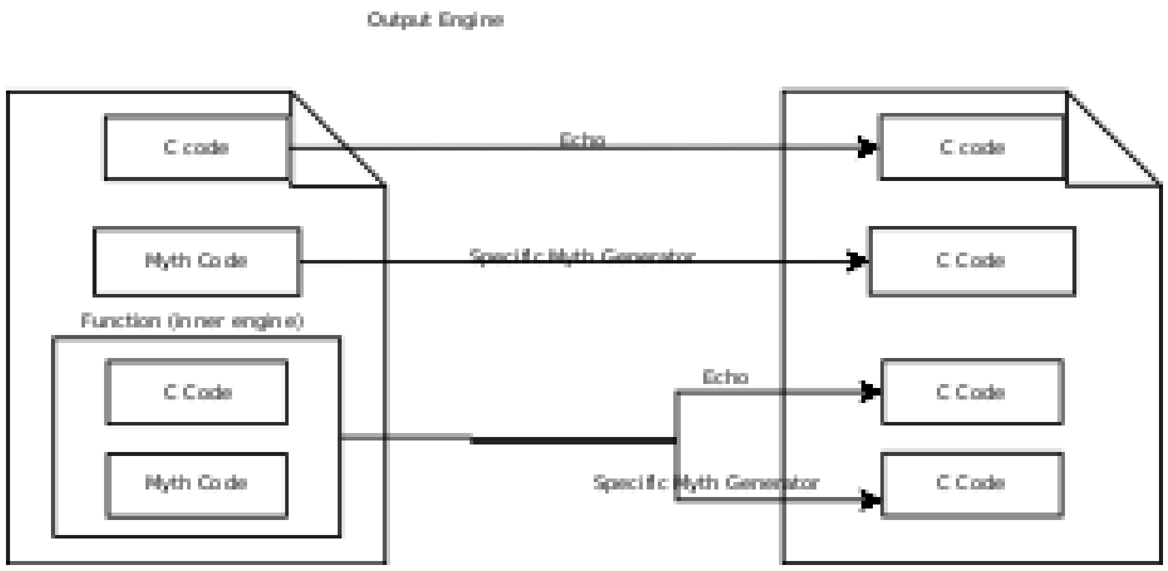
Instead of creating a heavily recursive output system the Myth preprocessor instead builds very flat structures. All parsed data exists in a data structure of two levels of depth at the deepest level. The data, in both levels, is stored using a sorted linked list. The list is sorted on the locations where the data was extracted from the source code.

For the locations where Myth data was found translation into C source code is required. The code generator loop calls a `generate` function pointer corresponding to the type of Myth data that needs to be translated. The C generation for almost of all these types is very straightforward. Only for the case of the function body is it complex. A function body generation function is complex because it can require a cast of a `self` pointer as well as contain interface calls or tuple return jumps that require further translation.

3.5.2. Function Body Output Engine

The function body engine is similar to the main output engine. It iterates over an ordered list of items. Each item contains the location from where the second item originated in the source code. In the case that the function body was defined inside a module a cast of the `self` pointer is required. This is inserted as the first statement following the opening brace of the function. Surrounding the `self` cast are `#line` directives.

It is very important that the line numbering is preserved in the output file. If it is not then the `#line` directives do no good and debugging and compiling become very difficult. The function body output engine examines the source location of adjacent output items and outputs the appropriate number of line feeds. This is necessary because the parser does not care about, nor does it capture, white space. The output from the preprocessor must preserve the line feeds. The preservation of other white space is not required and thus is not strictly preserved.



3.6. Myth to C Translations

Most source code in a Myth program is not modified by mythpp since it is mostly C code. Myth specific extensions must be collected and translated to C.

3.6.1. Bitsets Translation

The translation from a Myth bitset to a C source code is very straightforward. A Myth bitset is translated into an `enum`. Each bitset member is assigned a value according to how the bitset was defined. In addition each bitset member is prefixed with the name of the bitset. The following code lists a Myth bitset and the corresponding C code that it is translated to.

```
/* Myth code */
bitset exampleSet
{
        bitZero;
        bitOne;
        lowestByte: 0..7;
        bitTwo;
        /* The order of the ranges does not matter */
        byteThreeAndByteZero: 31..24,0..7;
}
```

```
/* C code */
enum exampleSet {
exampleSet_bitZero = (1 << 0),
exampleSet_bitOne = (1 << 1),
exampleSet_lowestByte = 0x000000ff,
exampleSet_bitTwo = (1 << 2),
exampleSet_byteThreeAndByteZero = 0xff0000ff
};
```

3.6.2. Tuple Return Translation

There are three types of tuple return syntax that require translation into C. A function declaration or definition that uses a tuple return must be translated into C. Returning a tuple from a function also requires translation. Finally, when a function is called that returns a tuple translation must be done for the assignment.

For the case of a function declaration of definition the types listed in the tuple return section must be listed in the parameter section of the function but as pointers to those types. Names for the new parameters are generated with the prefix `tuple_x` where `x` is a value that increases starting from 0. For the case of a function definition an identical translation is performed.

When returning a tuple value from a function translation must occur. In the context of a return each tuple parameter pointer is dereferenced and each expression in the tuple is assigned.

When making an assignment from a tuple function translation must also occur. This translation moves each identifier in the tuple into the beginning of the parameter list passed into the function. In addition each one of those identifiers is prefixed with a ’&’ to pass a pointer to that data. This is call-by-reference.

The following example shows a simple program that uses each type of tuple syntax. The example following that shows the C code that it is translated into.

```
/* Myth code */
tuple(int,int,int) function(int input);

tuple(int,int,int) function(int input)
{
        int x = 4;
        return tuple(x, x+1, 5);
}

int main(int argc, char *argv[])
{
        int x = 0;
        int y = 0;
        int z = 0;

        tuple(x,y,z) = function(5);
        return 0;
}
```

```
/* C code */
void function(int *tuple_0, int *tuple_1, int *tuple_2, int input);

void function(int *tuple_0, int *tuple_1, int *tuple_2, int input)
{
        int x = 4;
        *tuple_0 = x;
        *tuple_1 = x+1;
        *tuple_2 = 5;
        return;
}

int main(int argc, char *argv[])
{
        int x = 0;
        int y = 0;
        int z = 0;

        function(&x, &y, &z, 5);
        return 0;
}
```

3.6.3. Interface Translation

Translating a Myth interface declaration to C is a straightforward process. The only declarations allowed in an interface are function declarations. The C translation involves making a `struct`. The `struct` is named with the name of the interface and wrapped in a typedef also with the name of the interface. Inside the `struct` the first item is always a `void` pointer named `_self`. Then, each function declaration is translated into a corresponding function pointer declaration of the same name of the function declaration.

The name of `_self` was chosen instead of `self` to avoid a naming conflict inside module functions. Inside module function bodies `self` is used to access module instance data. To accomplish this the `void` pointer that is passed into the module function is cast to the type of the module. Examples of this can be seen in the Module Translation section.

Pointers were used for the `_self` and for the functions in an interface because an interface is intended to reference a module instance. Interfaces in Myth are not intended to be useful on their own. The data in the interface instance requires initialization which can be done by mythpp. It is not possible for mythpp to know when to initialize an interface instance and therefore it is required that interfaces are explicitly initialized in Myth.

```
/* Myth code */
interface foo{
        void bar();
        int rum(char coke);
        tuple(int,int) baz();
}
```

```
typedef struct foo {
        void *_self;
        void (*baz)(int *tuple_0, int *tuple_1, void *_self);
        int  (*rum)(void *_self, char coke);
        void  (*bar)(void *_self);
} foo;
```

3.6.4. Module Translation

Myth allows modules declarations and definitions. A module may have a list of interfaces that it implements. The list of interfaces that a module implements determines what code is generated. In the case of a module declaration it is not necessary to declare functions that are declared in an interface that is listed in the modules implements list. The Myth preprocessor also inserts function declarations for the interface initializer functions that the module definition translation automatically creates. For each function in a module the parameter list is modified. If tuples are used the tuple parameter pointers are listed first. Following any tuple parameter pointers is the `void` pointer to the module instance named `_self`. Finally the original function parameters are listed. When the functions are declared they are given a prefix name corresponding to the name of the module. In addition to declaring a set of module functions a `struct` that corresponds to the modules `self` type is generated. This `struct` is wrapped in a `typedef` and named after the module.

A module definition contains function definitions and optionally has an interface implementation list. For each function defined in the module the parameter list is modified according to the same rules that are used for creating the function declaration. Also a local variable declaration is inserted as the first line in each function body. This declaration creates a `self` pointer of the correct module type and is initialized with the `_self` `void` pointer that is passed into the function. This provides access to module instance data inside the function in a logical manner. For each module that is listed in the interface implementation list an interface initializer function is created. The initializer function is used to initialize an interface instance to a particular module instance.

The following example shows the translation of a module declaration and definition using a single interface.

```
/* Declare an interface named foo with a few functions */
interface foo{
        void bar();
        int rum(int coke);
        tuple(int,int) baz();
}


/* Declare a module named stumbles that implements the foo interface.
   Declare a self section that has some state.  Also declare a
   function (init) that is not part of the foo interface. */
module stumbles implements foo
{
    /* self */
    {
        int isDouble;
        int useTopShelf;
        int occupiedStools;
    }

    void init(int isDouble, int useTopShelf);
}

/* Define the module. */
module stumbles implements foo
{
    void init(int isDouble, int useTopShelf){
        self->isDouble = isDouble;
        self->useTopShelf = useTopShelf;
        self->occupiedStools = 0;
    }

    void bar(){
        self->occupiedStools++;
    }

    int rum(int coke){
        int rumType = 0;
        if(self->useTopShelf){
            rumType = 1;
        }

        return rumType * (self->isDouble ? 2 : 1) + coke;
    }

    tuple(int,int) baz(){
        return tuple(4,5);
    }
}
```

```
/* The C code that is generated from the Myth code above follows.  The
   line directives have been stripped as they clutter the output. */

/* Declare an interface named foo with a few functions */
typedef struct foo {
    void *_self;
    void (*baz)(int *tuple_0, int *tuple_1, void *_self);
    int  (*rum)(void *_self, int coke);
    void  (*bar)(void *_self);
} foo;


/* Declare a module named stumbles that implements the foo interface.
   Declare a self section that has some state.  Also declare a
   function (init) that is not part of the foo interface. */

void stumbles_baz(int *tuple_0, int *tuple_1, void *_self);

int  stumbles_rum(void *_self, int coke);

void  stumbles_bar(void *_self);

typedef struct stumbles {
    int isDouble;
    int useTopShelf;
    int occupiedStools;
} stumbles;

void  stumbles_init(void *_self, int isDouble, int useTopShelf);

/* NOTE: this is the initializer forward declaration.  This is
   automatically generated by mythpp */
foo *stumbles_foo(stumbles *in, foo *out);

/* Define the module named stumbles */
void stumbles_baz(int *tuple_0, int *tuple_1, void *_self){
    stumbles *self = _self;

    *tuple_0 = 4;
    *tuple_1 = 5;
    return;
}


int  stumbles_rum(void *_self, int coke)
{
    int rumType = 0;
    stumbles *self = _self;
    if(self->useTopShelf) {
        rumType = 1;
    }

    return rumType * (self->isDouble ? 2 : 1) + coke;
}



void  stumbles_bar(void *_self)
{
    stumbles *self = _self;
    self->occupiedStools++;
}



void  stumbles_init(void *_self, int isDouble, int useTopShelf)
{
    stumbles *self = _self;

    self->isDouble = isDouble;
    self->useTopShelf = useTopShelf;
    self->occupiedStools = 0;
}



foo *stumbles_foo(stumbles *in, foo *out){
    out->_self = in;
    out->baz = stumbles_baz;
    out->rum = stumbles_rum;
    out->bar = stumbles_bar;
    return out;
}
```

4. Porting Mythpp to Myth

The porting of the Myth preprocessor, mythpp, to the Myth language was an even more valuable experience than writing mythpp itself. The port exposed a lot of scenarios that were not explicitly tested. It revealed a number of serious bugs. In addition to making mythpp a more usable program it also provided the opportunity to test most of the Myth language features in a real program.

The porting took place in a series of steps. Each step increased the number of things that had to work correctly in mythpp.

4.1. Phase 1 - Unmodified Code

The first step was to build mythpp simply by running the unmodified C source code of mythpp through mythpp. Since Myth is a superset of C the result should be a program that is identical to the original.

The first issue that was discovered was that it is necessary for Myth source files to be run through the C preprocessor prior to being run through mythpp. The reason for this is because of typedefs. For example when `stdio.h` is included the type FILE is declared. Without including `stdio.h` `FILE` is parsed as an identifier and not a type name. This results in a syntax error from mythpp.

The second issue that was discovered was that the handling of typedefs was not correct. The parser specification and lexical specification from Jeff Lee left some of the details of parsing typedefs unfinished. The first attempt at an implementation did not cover all of the possible `typedef` cases. This became evident as soon as any standard include files were included. The standard include files use typedefs in a wide variety of ways.

Finally this phase revealed an issue related to using the GCC C compiler. GCC has a builtin type for handling variable argument lists. Since it is builtin there are no typedefs that define it. It was necessary to add a special test in the lexical specification to scan `__builtin_va_list` as a `TYPE_NAME` instead of an `IDENTIFIER`.

Once these bugs were corrected mythpp could compile itself as long as no Myth extensions were used. This was a huge milestone because a fair number of standard header files are used in mythpp. Parsing those was a great test of the mythpp parser.

4.2. Phase 2 - Use Tuple Returns

The next least invasive step in porting mythpp to Myth was the addition of tuple returns. There was one compilation unit, the `mythFile` unit that benefited from the use of tuples. This unit is a wrapper around a handful of file operations. Most of these operations can fail and report error information via the global `errno` variable. These functions were modified to return the desired value as well as an indication of success and an error code.

This step revealed a serious oversight in designing mythpp. The original plan was to put all tuple parameters as the last parameters for a function. Shortly into the project it was realized that this would not work for variable argument functions. The next choice was to place the tuple functions as the first parameters in a normal function call, or immediately following the self for a function defined in a module. During this phase of the port it became evident that this approach was incorrect as well.

The problem is that a module function call is not translated in any way by mythpp. To the parser a module function call is a normal function call. The programmer has to manually pass in a pointer to the module instance for any module function call. It is only for interface calls that a `self` pointer is passed in automatically. If a module function uses tuples then, at the pointer where it is called, mythpp does not know that the first parameter is the `self` pointer and must be passed as the first parameter. To the parser the `self` pointer is the same as any other parameter and thus is passed to the function after the tuple arguments. The solution to this problem was to make tuple parameters be the very first parameters in a function under any condition. Luckily this did not require significant code changes.

4.3. Phase 3 - Convert to Modules

This phase, although very time consuming, went very smoothly. It took a lot of time to implement because a naming scheme of appending "_t" for type names is used in the C version of mythpp. Mythpp on the other hand creates a type name that corresponds exactly to the name of the module. For this reason all of the instances of types with a name suffix of "_t" had to be stripped of their suffix as they were ported.

In addition to the change of names for module types it was found that some poor choices for module names were used. In particular there were modules named `bitset`, `interface`, and `module` in the original mythpp source code. These are reserved words in Myth and thus caused syntax errors when passed through mythpp.

4.4. Phase 4 - Convert generator_t to Interface

The fourth phase of the port also went surprisingly well. In the C version of mythpp there is a `generator_t` type. This type has a `void*` and a function pointer that points to a function to generate C code. This data type is used heavily in the C version of mythpp because it allows modules of different types to be worked with as the same type. This sort of code is exactly what a Myth interface is designed for. The changes for this phase included adding `implements` lists to existing modules and changing `generator_t` instances to be initialized using Myth initializers. No bugs where found during this phase.

In one instance some creative coding was done to avoid a sizable change to the original mythpp code. The module for modules actually has two C code generation functions. One is for generating a module declaration and one is for generating a module definition. The code and data required for the two types are nearly identical except in the way that output is generated. In the original source, since function pointers were manually assigned for the `generator_t` instances, the `generateDecl` function could be assigned to the `generate` function pointer of the `generator_t` type just as easily as the `generate` function could be assigned to that variable. Since mythpp generates initialization code there was no clean way to achieve the same thing. The approach that was taken was to manually reassign the generate function pointer after calling the Myth interface initializer. While this sort of breaks an implicit rule that the programmer should not need to know the structure of an interface it was a benefit in this particular situation.

4.5. Line Directive

At first it was believed that mythpp could exist without generating `#line` directives in the output code. During the porting effort it became clear that mythpp wasn't a viable tool without `#line` directives. The line directives instruct the compiler where a line of code originated from. C compilers have this directive because the source they compile has been changed by the preprocessor. The C compiler needs to provide error messages that relate back to the true source file.

To add this feature required significant work. The parser had to track the line directives that were in the code. To make matters more challenging the GNU C preprocessor uses a non-standard format for the line directive. Changes were made to the parser as well as the addition of optional command line switches that allow the specification of the true source file. This switch is necessary because mythpp must operate on files already pre-processed by the C preprocessor.

The benefits of adding this feature were greater than anticipated. The feature was added so that errors reported from the C compiler would reference their true origin. What was not anticipated was that the GNU debugger understands this information as well. A crash bug was intentionally inserted temporarily into the myth code. The ported mythpp binary was run under the GNU debugger, gdb. When the program crashed gdb was able to walk stack frames back to the exact line in the Myth source code that caused the crash, not the line from the file that was ultimately compiled.

4.6. Bitsets in the Port

The C version of the Myth preprocessor made no use of bit operations and therefore the port made no use of Myth bitsets. It would have been artificial to insert them into the port.

5. Where Myth is Successful

Porting the C version of mythpp to the Myth programming language provided an excellent opportunity to explore the Myth programming language. Overall Myth seems to be a benefit. Current implementation restrictions made some tasks annoying as they clashed with typical C behavior. In general I feel that the Myth version of the Myth preprocessor is more readable than the C version.

Had mythpp been written in Myth from the beginning the modules syntax would have saved a significant amount of typing. More importantly the use of modules conveys a more structured approach to the organization of the source code. Modules made clear the intent that certain data is associated with a particular set of functions while not limiting access to the data in any way.

Interfaces proved to be a success. In the mythpp port the code that deals with accumulating data destined for the output engine is much clearer than that in the C version. In the C version many mental steps are required to understand the intent of the `generator_t` type. In Myth the intent of the generator interface is evident with far fewer mental steps. This makes the code easier to understand and thus more maintainable.

A major goal of Myth is to promote easily readable source code. The required `@` character for tagging interface calls enhances the readability of a program. It alerts the reader that the call is not a direct function call and that more mental work is required to determine what function will actually be called. Languages like Java and C++ make it difficult to understand what function will be called without a detailed examination of a large amount of source code. Determing the destination of a function call in C++ is a complicated task. A C++ code reader must ask multiple questions about the symbol. They first must ask if the function is defined in the current class. If it is not they must then ask if the function is defined in any class in the inheritance chain. To complicate matters further a function in these languages can be defined with the same name in the current class and in any class in the inheritance chain. Further the call behavior will vary depending on how the function was defined in the current class. In fact, with dynamic binding, it can be impossible to know what code will be executed until runtime.

The `mythFile` module wraps a few system calls. The module provides utility functions for file operations. Some operations including opening, memory mapping, and unmapping files. Nearly every file operation can result in an error. Extended error information is reported from the system via the global errno variable. Myth tuple returns made for a cleaner syntax when dealing with these sorts of errors.

In this implementation of Myth a caller can not ignore the return value of a function that uses tuples. This was a design compromise. In the case of mythpp this proved to be a beneficial restriction. However, the general case is that this is not desirable. This sort of behavior clashes with normal C and feels unnatural. The intent of tuple returns was to make returning multiple values clear. In this goal Myth is a success.

6. Limitations of Mythpp

As a language Myth is successful in its goals. The mythpp application has many implementation limitations.

6.1. Tuple Returns

In the current implementation of mythpp it is awkward to deal with returning a `void*` when using a tuple return. Doing so will generate a warning from the compiler due to a type mismatch. C allows for the assignment of `void*` to any other pointer type without casting. Given a tuple return such as `tuple(void*, int)` it would be expected that the following code would be legal.

```
tuple(void*, int) exampleFunction();

char *example = malloc(1024);
int x = 0;
tuple(example, x) = exampleFunction();
```

This generates a warning because the actual C output is:

```
void exampleFunction(void **tuple_0, int *tuple_1);
char *example = malloc(1024);
int x = 0;
exampleFunction(&example, &x);
```

In this example the variable `example` is not a `void**` it is a `char**`. The `void**` type doesn't have the same sort of relaxed rules that the `void*` type does and a warning is generated.

In C it is perfectly valid to ignore the value returned from a function. This can be very convenient. Due to the way tuple returns are implemented in mythpp it is not possible to return the result of a function that was declared to return a tuple. This is because the tuple return values are translated into "out" parameters for the function. If a tuple return function is called with out examining the returns values the C compiler will generate an error because too few parameters are passed to the function.

```
tuple(int,int) tupleFunction(int input);
tuple(x,y) = tupleFunction(4);
tupleFunction(5);
```

The Myth code above is translated into:

```
void tupleFunction(int *tuple_0, int *tuple_1, int input);
tupleFunction(&x, &y, 4);
tupleFunction(4);
```

Clearly it is not legal to call `tupleFunction` with a single argument. As a possible workaround dummy variables can be used for `x` and `y`. That is very inconvenient and wastes stack space.

For the same reason that tuple return values can not be ignored a tuple can not be returned directly from a function. This is inconvenient, leads to sloppy code, and clashes with normal C behavior.

```
tuple(int,int) tupleFunction1(){
        return tuple(4,5);
}
```

```
tuple(int,int) tupleFunction2(){
        return tupleFunction1();
}
```

The example above is perfectly legal Myth syntax. However when sent to the C compiler this will generate an error. In `tupleFunction2` the limitation of ignoring a tuple return is seen. In this case it is not obvious that the values of the tuple return are being ignored. The C output makes it clear why this is an error.

```
void tupleFunction1(int *tuple_0, int *tuple_1){
        *tuple_0 = 4;
        *tuple_1 = 5;
        return;
}

void tupleFunction2(int *tuple_0, int *tuple_1){
        return tupleFunction1();
}
```

This generates an error because not enough parameters are passed into tupleFunction1. To work around this implementation limitation temporary variables must used to store the intermediate result. The temporary variables must then be used in the return statement. The workaround forces needless copying. The C output for the workaround illustrates this problem very clearly.

```
/* Myth Program Workaround */
tuple(int,int) tupleFunction1(){
        return tuple(4,5);
}

tuple(int,int) tupleFunction2(){
        int tmp1;
        int tmp2;
        tuple(tmp1, tmp2) = tupleFunction1();
        return tuple(tmp1, tmp2);
}

void tupleFunction1(int *tuple_0, int *tuple_1){
        *tuple_0 = 4;
        *tuple_1 = 5;
        return;
}

void tupleFunction2(int *tuple_0, int *tuple_1){
        int tmp1;
        int tmp2;

        tupleFunction1(&tmp1, &tmp2);
        *tuple_0 = tmp1;
        *tuple_1 = tmp2;
        return;
}
```

6.2. Interfacing with other Libraries

During the porting process an attempt was made for an interface that allowed for the comparison of any two like modules that implemented that interface. The intent was to use this interface with the `sortedList` module. The `sortedList` module is a wrapper around the `glib GSList` type. A `GSList` is a singly linked list that has support for inserting and maintaining a sorted order. The call for inserting an item in sorted order takes a function pointer as a parameter. The first problem was that the signature for the comparison function passed two `const void pointers`. Myth doesn't have a mechanism for having the `self` pointer being `const` and so a cast discarding the `const` would be required. The second problem was found while implementing the comparison function in a module. Since glib knows nothing about interface calls an interface pointer is passed to the function, not a pointer to the `self`. This will not generate a warning from the C compiler because the `self` pointer is always declared in module functions as a `void` pointer. Casting should work under these conditions because the `self` pointer is the first item in an interface `struct`. However, that is somewhat of kludge and requires knowledge of the internals of interface structures.

6.3. Interfacing with GNU GCC C compiler

When using the GNU C preprocessor GNU specific line directives are inserted into the code. The Myth preprocessor was modified to understand these directives. The Myth preprocessor only outputs the standard syntax however. The final input arriving to gcc has been pre-processed. If the -pedantic flag is used gcc will warn that GNU specific line directives exist in the code. This effectively makes the -pedantic flag useless as a single file typically contains large quantities of line directives inserted by the C preprocessor.

7. Experiences

Implementing Myth forced careful thought into every aspect of each language feature. This brought to light many shortcomings from the original plan that needed to be modified.

7.1. Tuples

Originally Myth tuples were to be a new C data type. After some thought it was decided that it was not desirable to make tuples a data type. In addition it was found that implementing tuples as a true data type would be very difficult to achieve using a preprocessor. The original idea was to implement tuples using `structs`. The types in the Myth tuple declaration would be used to create the `struct` declaration. This technique has many problems though. For one thing the caller of a function would need to know about the `struct` that was returned. But the types can't be inferred at the caller unless identifiers are tracked according to scoping rules. If the syntax of `tuple(...) = tupleFunction();` were disallowed and instead required the caller to declare a `tuple(...)` variable to be assigned to the output of the function then it would be possible. That seems like a good solution except that extracting the data from that tuple variable would be cumbersome because the item names in the generated structure are automatically generated. Finally, offering a tuple as a data type would lead to the possibility of a recursively defined tuple. That sort of situation would be difficult to deal with when generating proper C output.

Another possible way to implement tuple returns was to generate a `struct` definition where the function is declared. This approach would correct the limitation of ignoring return values from a function. Identifier names for structure elements could be easily generated using a numbering scheme with some prefix. This technique however has a number of challenges. First, care must be taken not to declare the same structure twice. This could be done by having a registry to lookup the name of an already declared `struct`. Inside the function body a variable of the specified return type would have to be declared. When calling a function that returns a tuple the Myth preprocessor could automatically generate a variable of the correct structure type for the return and then assign each item of the structure to the variables in the `tuple()`. This has two problems. First, it results in twice the number of copies that are really necessary. The other problem is that, unless C99 code is generated, variables can't be declared in the middle of a function. Since one goal of this implementation was to be self hosting, and because the basis of the preprocessor is a C89 type grammar specification, this choice was not feasible.

The approach of using "out" parameters proved to be the most reasonable solution to this problem. It is simple to rewrite function declarations and tuple assignments to the correct C code. No unnecessary copying is performed. Also, it is not required to keep track of the scope of identifiers in any way. The primary disadvantage to this approach is that it becomes impossible to ignore the return value of a tuple function. For some coding styles that may be desirable but it clashes with normal C behavior. A less severe disadvantage to this approach is that dealing with void pointers returned via a tuple is not intuitive. Type mismatch warnings are generated by the C compiler where the Myth programmer wouldn't expect to see a warning.

7.2. Interfaces

Originally it was planned to implement interfaces as a set of function pointers contained within a module. The set would be a `struct` so that it could be referenced independently of a module as well as to easily access interface functions from a module. Upon closer examination of dealing with `self` pointers it became evident that the approach could not work. If a function was declared that took an interface as a parameter there would be no way to carry the `self` pointer around with it. It became evident that an interface had to have a way to find the correct `self` pointer to be passed during a function call.

To facilitate correct passing of the `self` pointer interface declarations were removed from module declarations. This left a module as a single `struct` that is the state of the module. Also a `void*` was added to all interface structures. Once an interface structure is initialized this item will point to the correct module state.

The breaking of interfaces out into their own entities also brought to light the fact that interface structures need to be initialized prior to use. This concept does not clash with normal C code as all variables are uninitialized by default. However, it would be unreasonable to force a Myth programmer to manually assign the `self` pointer and every function pointer for an interface prior to use. The Myth preprocessor collects enough information that this initialization code can automatically be generated. The Myth preprocessor will generate an initialization function for every interface that a module is declared to implement.

It was originally planned that the Myth preprocessor would track identifier names and types so that it could detect when a function call was made using an interface and rewrite the source code to pass in the `self` pointer to the function. While possible this is a difficult task. Scope rules are one challenge. Another challenge is determining type. A function could return a structure that contains an interface. It would be very difficult to determine that the following is an interface call:

```
returns Struct().interfaceInstance->function();
```

In addition to being difficult to track types it would not be possible to pass in a `self` pointer with out causing a side effect. In the example above the only way the preprocessor could pass the self pointer would be to rewrite it as

```
returns Struct().interfaceInstance->function(
                        returns Struct().interfaceInstance->_self);
```

Clearly that can result in unexpected conditions when `returns Struct()` has a side effect.

To solve the problem of identifying interface calls it was decided to tag interface calls. By tagging an interface call the programmer tells the preprocessor that the function call is a call to an interface and will require a `self` pointer. The preprocessor can then easily generate code to reference the `self` pointer. This technique does not guard against side effects however. To guard against side effects a requirement is made that interface calls may only be made on an identifier. The interface call can be accessed via either a `.` or a `->` depending on if the identifier refers to a pointer or an actual structure. Requiring an identifier is inconvenient to the programmer but seemed to be the best solution given all of the constraints.

8. Future Work

There are many areas where mythpp could be improved.

8.1. Remove Identifier Restriction

Currently it is required that interface calls be made on an identifier. This results in the use of dummy variables and needless copying. It would be ideal if interface calls could be made on any interface value, even one that is not directly assigned to an identifier. It would be possible to remove this restriction if the scope and type of identifiers were tracked. Another possible solution would be to create a Myth compiler (as opposed to a preprocessor).

8.2. Add C99 Features

C99 makes a few additions that would be good additions to Myth. With C99 variables can be declared at any point in a function. It is not required that variables be declared at the beginning. Another C99 feature that would a good addition to Myth is dynamic arrays. This sort of change would require some changes to the parser specification and require a C99 compliant C compiler, or some very sophisticating processing to translate C99 input to C89 output.

8.3. Implement as GCC Front End

A lot of the limitations that exist in the current implementation of mythpp arise due to design compromises chosen because it was felt to be too difficult to implement Myth as a GCC front end. GCC version 4.0 was released on April 20th, 2005, just a few days after the completion of mythpp. This new version of GCC makes writing compiler front ends easier because a front end programmer need only to build a tree. In the past the front end was required to provide RTL (register transfer language) instructions to the back end. [tromey]

8.4. More Robust Semantic Analysis

Currently the Myth preprocessor does very little semantic analysis. One reason for this is that the majority of the semantic analysis is provided by the back end C compiler. The implementation for bitsets is one area where more semantic analysis would be a good addition. Bitsets are currently emitted as enums. Enums however are handled by C compilers as the `int` type. Mythpp does no checking to see if there are enough bits in an int to satisfy the programmers request. At first glance this sort of check seems to be a simple addition. However, when it is considered that mythpp may not be run on the target platform the problem of determining the maximum size of an `enum` becomes more difficult.

8.5. Add const Feature

In some cases it is desirable to specify that the `self` pointer for a module or interface function be declared as `const`. Currently the Myth language has no provision for that. A potential change to Myth would be to allow the `const` keyword following a function declaration when declared in an interface or module.

8.6. Allow Ignored Tuple Parameters

The current implementation for tuples returns is awkward to use. It would be great to see a better implementation for tuple returns implemented that would follow the customs of C. It would also be nice if it were possible to ignore individual items in a tuple return instead of an all or nothing approach. A new implementation should allow for tuple returns to be ignored and for a function to return a tuple that was returned from some other function. At the same time it would be desirable to avoid making tuples a data type.

9. Conclusion

The Myth programming language succeeds in the tasks it was designed to accomplish. Modules ease the burden of rewriting prefix names over and over. Interfaces provide a simple and clear mechanism for writing reusable algorithms. The `@` tag on interface calls is a benefit by drawing attention to the fact that the call is not a direct call. Bitsets make declaring and maintaining groups of related bit masks much easier.

Porting the Myth preprocessor to itself highlighted the successes of the programming language but also highlighted the deficiencies of the Myth preprocessor. The Myth preprocessor that was developed is sufficient as a learning tool and for testing the viability of the Myth programming language.

The project as a whole was a success. I was able to learn about programming languages and to test out my ideas for a new programming language. I learned about the capabilities and limitations of compiler generator tools. I was able to implement a usable preprocessor to explore the usability of the Myth programming language.

References

[synder] Alan Snyder, *Encapsulation and Inheritance in Object-Oriented Programming Languages*, ACM Press, 1986, 0-89791-204-7, 38-45.

[tromey] Tom Tromey, *Writing a GCC Front End*, Specialized Systems Consultants, Inc., May, 2005, 78-81, Volume 2005, 133.

[kernighan-plauger] Brian Kernighan and P. J. Plauger, *Software Tools.*, Addison-Wesley, 1976, 201-03669-X.

[raymond] Eric S. Raymond, *The Art of Unix Programming*, Addison-Wesley, September, 2003, 0131429019.

[lee] Jeff Lee, *Ansi C grammar and lexical specification*.

ftp://ftp.uu.net/usenet/net.sources/ansi.c.grammar.Z

[fedor] *GNUStep Programmers Manual*, Adam Fedor.

http://www.gnustep.org/resources/documentation/Developer/Base/ProgrammingManual/manual_toc.html#SEC_

[richie] *The Limbo Programming Language*, Dennis M. Richie.

http://www.vitanuova.com/inferno/papers/limbo.html

[davis] *A Guide to the S-Lang Language*, John E. Davis.

http://www.s-lang.org/doc/html/slang.html

[stroustrup] Bjarne Stroustrup.

http://www.research.att.com/~bs/bs_faq.html#bootstrapping

[gcc] *GCC Manual*, Free Software Foundation, Inc..

http://gcc.gnu.org/onlinedocs/gcc-4.0.0/gcc/Submodel-Options.html#Submodel-Options

[glib] *Glib Homepage*.

http://www.gtk.org

[boehm-demers-weiser] *A Garbage Collector for C and C++*, Hans J. Boehm, Alan Demers, and Mark Weiser.

http://www.hpl.hp.com/personal/Hans_Boehm/gc/

A. Appendix

A.1. Partial Myth Grammar

This is the Myth specific pieces of the grammar used for mythpp. Non-terminals are in lower case while terminals are in upper case. Non-terminals that do not appear in this listing are based on Jeff Lee's C grammar [lee].

To simplify the grammar for reading purposes a few conventions are used. If a non-terminal has a `_OPT` suffix, then that non-terminal is optional.

Non-terminals ending with `_LIST` denote a list of that non-terminal separated by whitespace.

Non-terminals ending with `_CLIST` denote a comma separated list.

Non-terminals ending withe `_SLIT` denote a semi-colon separated list.

```
tuple_id
: TUPLE ’(’ identifier_l ’)’

module_declaration_specifiers
: type_specifier module_declaration_specifiers_OPT

tuple_assignment_fcall
: IDENTIFIER ’(’ argument_expr_l_OPT ’)’
| call
```

```
tuple_assignment_statement
: tuple_id assignment_operator tuple_assignment_fcall

call_tail
: ’(’ argument_expr_l_OPT ’)’

arrow_or_dot
: PTR_OP
| ’.’


call
: ’@’ IDENTIFIER arrow_or_dot IDENTIFIER call_tail


statement
: simple_statement
| jump_statement
| tuple_assignment_statement
| call
| call_assignment_statement


call_assignment_statement
: IDENTIFIER assignment_operator call


jump_statement
: normal_jump
| RETURN TUPLE ’(’ argument_expr_l ’)’ ’;’


external_definition
: function_definition
| interface_decl
| module_decl
| module_def
| bitset_decl
| declaration
| tuple_function_declaration


tuple_function_declaration
: basic_tuple_decl ’;’
| declaration_specifiers basic_tuple_decl ’;’


basic_tuple_decl
: tuple_type IDENTIFIER function_params

module_def
```

```
: storage_class_specifiers_OPT basic_module_def


basic_module_def
: MODULE id_or_typename '{' module_function_def_LIST '}'
| MODULE id_or_typename IMPLEMENTS id_or_typename_l '{' module_function_def_LIST '}'

module_decl
: storage_class_specifiers_OPT basic_module_decl

basic_module_decl
: MODULE IDENTIFIER  module_decl_body
| MODULE IDENTIFIER IMPLEMENTS id_or_typename_l module_decl_body

module_decl_body
: '{' self_decl module_function_decls_OPT '}'
| '{'  module_function_decls self_decl '}'


self_decl
| '{' struct_declaration_l_OPT '}'

bitset_decl
: BITSET IDENTIFIER '{' bitset_member_LIST '}'


bitset_member
: IDENTIFIER ';'
| IDENTIFIER ':' range_item_CLIST ';'


range_item
: CONSTANT RANGE_SEPERATOR CONSTANT

tuple_type
: TUPLE '(' parameter_type_l ')'


interface_decl
: INTERFACE IDENTIFIER '{' module_function_decl_SLIST '}'


module_function_decl
: basic_tuple_decl
| function_declarator function_params
| module_declaration_specifiers function_declarator function_params


module_function_def
: basic_tuple_decl function_body
| function_declarator function_params function_body
| module_declaration_specifiers function_declarator function_params function_body
```