

Filet-O-Fish Tutorial: The Fugu Error Definition Language

Pierre-Evariste DAGAND

Introduction

In this tutorial, we aim at illustrating the usage of Filet-O-Fish (FoF). Therefore, we will implement a small Domain-Specific Language (DSL), using FoF in the back-end. However, we will also cover a broader topic: how to design a new DSL taking advantage of FoF particularities. In particular, this DSL will be an Error Definition Language (EDL) called Fugu. Its functionalities are described in Section 1.

Hence, we will adopt a step-by-step approach. While covering the various phases of development, we will try to devise a more “principled” approach, which could serve for future development.

One of our *partis pris* is that such work should start from mature C code. FoF is meant as a safe meta-language, allowing the DSL designer to, first, abstract over C and, then, manipulate these high-level constructions in a powerful environment, such as Haskell or a theorem prover.

For the sake of brevity, we will focus our presentation on the DSL back-end. Hence, we will not write a parser for Fugu: the syntax of this small language will be *embedded* in Haskell, by the means of some *combinators*. We will take care of avoiding any confusion between all those languages.

This tutorial is organized as follow. In a first Section, we specify our requirements concerning Fugu. In a second Section, we consider define a small class of errors and describe how we would like to compile them down. From there, we carry a step-by-step implementation of Fugu in Section 3.

1 An Error Definition System

The need for an Error Definition System arises from the deficiencies of the traditional scheme, namely the `errno.h`, or similar, file. In this file, developers typically aggregate the error codes of all errors that could potentially occur in the system. In order to scale, they quickly “overload” the meaning of these error codes: instead of defining distinct error codes, they use a single code which has multiple causes. For example, the `EINVAL` error of FreeBSD signals that “some invalid argument was supplied: for example, specifying an undefined signal to a signal function or a kill system call”. In parallel, they also abuse the return values of functions. For instance, a function returning a `NULL` pointer is typically signalling an error. However, this does not provide much information to the developer facing such case.

Although this might have been historically relevant, there is no incentive, today, to limit the error code space to 255. Using 16 bits would allow us to define 65535 error codes. In such a huge space, we would be freed from the burden of overloading error codes and functions return values: when it is relevant, the developer should be able to define new error codes. Hence, he could handle errors more precisely, in the code as well as during the debugging process.

However, by imposing a flat name-space, the `errno.h` approach reduces the benefit of defining more precise error codes. When an error occurs, the developer is not able to relate it to a subsystem: it is simply defined as one error among thousands of others. In a sense, the error is more precise but still lacks some context. Being able to give a context to an error would be a step forward, notably during debugging.

Finally, it is often the case that an error in one function must be reported to several functions in the call-stack. Whereas the depth of the call-stack is generally low, it is extremely inconvenient to define case-

specific error codes at each level (the number of cases growing exponentially with the depth). Hence, we generally give up accuracy and report a global, overloaded error code.

For these reasons, we provide the developer with the power of defining classes of errors per “component”. The notion of *component* is purposely kept fuzzy: it is up to the developer to define its components, on a case-by-case basis. For example, it could be a single function or a class of related functions.

By using a 16 bits space, we are able get rid of the artificial limitation of the state-space. But we can also take advantage of this sub-word quantity in our 64 bits machines. Indeed, in one machine word, we are able to handle 4 error codes. Hence, we can build the call-trace by *pushing* an error code in the previous one, and so on along the call-path.

By relying on a tool to generate C code for us, we can also get rid of a lot of boilerplate code. For example, when defining an error code, we label it with a descriptive message as well as a short acronym. Then, given an error, we should be able to report it to the user, in a meaningful way.

Finally, in the event of wild write – such as a buffer overflow – on the error code variable, we would like the error code to become meaningless, instead of signaling an unrelated error. Therefore, we should use random numbers to identify errors. Hence, the common consequences of a wild write, such as overwriting with 0 or with a character, could be detected more easily.

2 C Sample Code

2.1 Starting from C

First of all, we would like to stress the importance of starting from a mature C code. The first and foremost reason is the ability to *discuss* this draft with the final users of the DSL: whereas not everyone is familiar with Haskell and FoF, C is the *lingua franca* for everyone. Thanks to this support, users are given the power to influence the design of the DSL, without having to handle its machinery. Moreover, this C file is an inexpensive support for testing and debugging: we can quickly modify its behavior without dealing with the compilation machinery.

Although disturbing at first, FoF should *not* be meant as a programming language. First and foremost, FoF gives a meaning, the semantics, to your DSL language. The fact that it can be compiled to C comes as a bonus.

Finally, a “good” DSL should be defined by an alternative semantics, which would allow the DSL designer to check the correctness of the compiler. Having a C sample of a particular instance can play this role, for this test-case. Although it does not help in obtaining formal guarantees, it is a first step toward a seemingly correct compiler.

2.2 Our Test-Case

In the following, we will define the error codes for two components: “system” and “pci”. The “`system_err`” class of errors will contain the following cases of success:

`SYS_OK` No error

`SYS_ILIA` Mafioso Pasta (private ETH joke, on SOSP lunches)

And the following cases of error:

`SYS_FAIL` Failed

`SYS_TEM` Kernel Hacker

`SYS_IPHUS` Analphabet Greek

The “`pci_err`” class of errors is more serious, with the following cases of success:

`PCI_OK` No PCI error

PCI_GET_CAP That is my cap

And this case of error:

PCI_CAP_NOK Lost my cap

In the next Section, we review the C code *we would like to generate* from this or a similar description. Bear in mind that this code is the result of a long and contradictory discussion with the final users.

2.3 The C Sample

2.3.1 Includes

First of all, we will need, more or less, the following includes:

```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdarg.h>
```

2.3.2 Type Definitions

We define an error value as 64 bits machine word. This encompass both the current error code and, potentially, 3 others stacked error codes.

```
typedef uint64_t errval_t;
```

2.3.3 Error codes

We define the error codes in an enumeration. Each code is identified by a random number between 0 and 65535. The success codes are identified by an odd number whereas failure codes are identified by an even number. This will come handy when we define a global `err_is_ok` and `err_is_fail` functions.

```
enum err_code {
    SYS_OK = 11,
    SYS_FAIL = 84,
    SYS_ILIA = 93,
    SYS_TEM = 2,
    SYS_IPHUS = 34,
    PCI_OK = 41,
    PCI_CAP_NOK = 28,
    PCI_GET_CAP = 16
};
```

2.3.4 Labels

Then, we statically define the previous error descriptions, acronyms, and domains. This way, we will be able to provide a descriptive notification to the user in case of error.

```
static char *err_domains[] = {
    "System_err", "Pci_err", "Undefined"
};
static char *err_codes[] = {
```

```

    "SYS_OK", "SYS_FAIL", "SYS_ILIA",
    "SYS_TEM", "SYS_IPHUS", "PCI_OK",
    "PCI_CAP_NOK", "PCI_GET_CAP",
    "UNDEFINED"
};
static char *err_msgs[] = {
    "No_error", "Failed",
    "Mafioso_Pasta",
    "Kernel_Hacker",
    "Alphabet_Greek",
    "No_PCI_error",
    "Lost_my_cap",
    "That_is_my_cap",
    "Undefined_error",
};

```

2.3.5 Getting and Stacking Errors

Given an `errval_t`, we will need to extract the latest error code. This is done by accessing the lowest 16 bits.

```

static inline enum err_code err_no(errval_t err){
    return (err & ((1 << 17) - 1));
}

```

Similarly, we can stack up an error code in an existing error value: we simply have to shift-left the stack and write the current error code in the lowest 16 bits.

```

static inline errval_t err_push(errval_t err, enum err_code code){
    return (err << 16) | code;
}

```

2.3.6 Success and Failure

We will frequently need to test an error code for failure or success. Hence, we provide two function which are both efficient and apply to any error code.

```

static inline bool err_is_ok(errval_t err){
    return err_no(err) % 2;
}
static inline bool err_is_fail(errval_t err){
    return 1 - (err_no(err) % 2);
}

```

2.3.7 Error-specific Tests

Then, we might need to test some error codes against a specific error. This is simply achieved by the following functions.

```

static inline bool err_is_sys_fail(errval_t e){
    return err_no(e) == SYS_FAIL;
}
...
static inline bool err_is_pci_ok(errval_t e){

```

```

    return err_no(e) == SYS_OK;
}
...

```

2.3.8 Asserting Success and Failure

Similarly, during the debugging phase, we might need to assert the correctness or incorrectness of a result. Hence, the following assertions.

```

static inline void err_assert_ok(errval_t err){
    assert(err_is_ok(err));
}
static inline void err_assert_fail(errval_t err){
    assert(err_is_fail(err));
}

```

2.3.9 Printing Errors

```

static inline char *err_getstring(errval_t err) {

    switch (err_no(err)) {
    case SYS_OK: {
        return err_msgs[0];
        break;
    }
    ...
    case PCI_OK: {
        return err_msgs[5];
        break;
    }
    ...
    default: {
        return err_msgs[8];
        break;
    }
    }
    return NULL;
}

```

```

static inline char *err_getcode(errval_t err ) {

    switch (err_no(err)) {
    case SYS_OK: {
        return err_codes[0];
        break;
    }
    ...
    case PCI_OK: {
        return err_codes[5];
        break;
    }
    }
}

```

```

...
default: {
    return err_codes[8];
    break;
}
}
return NULL;
}

static inline char *err_getdomain(errval_t err ) {

    switch (err_no(err)) {
    case SYS_OK: {
        return err_domains[0];
        break;
    }
    ...
    case PCI_OK: {
        return err_domains[1];
        break;
    }
    ...
    default: {
        return err_domains[2];
        break;
    }
    }
    }
    return NULL;
}

```

2.3.10 Printing the Call-trace

Finally, mostly during debugging, we would like to inspect the call-trace leading to an error. This is achieved by the following code. First, it verifies that the stack contains an error. Then, it *pops* error codes from it and prints their domain, description, and acronym.

```

static inline void err_print_calltrace(errval_t err){

    if (err_is_fail(err)){

        enum err_code x;
        while( (x = err_no(err)) != 0 ){
            printf("Failure:_(%12s)_%20s_[%s]\n",
                err_getdomain(x),
                err_getstring(x),
                err_getcode(x));
            err = err >> 16;
        }
    }
}

```

This concludes our draft implementation of an error management system. Now, we are aware of our needs, what the user needs to specify and what constructs should we abstract away. Let us start implementing Fugu, now.

3 Fugu: the Error Definition Language

In the following Sections, we describe the implementation of Fugu. As usual with me, you are reading a literate code: the Haskell compiler processes this very same file. Therefore, you are reading the real Fugu's code. In Section 3.1, we define Fugu's AST and a set of combinators, to be able to embed Fugu's language into Haskell. In Section 3.2, we implement the back-end using Filet-O-Fish.

3.1 Fugu Embedded Syntax

3.1.1 The Abstract-Syntax Tree

First, we define the Abstract-Syntax Tree (AST). An error definition file is a list of error classes:

```
type Errors = [ErrorClass]
```

Where an error class is identified by a string, its name, and a list of error definitions:

```
data ErrorClass = ErrorClass String [ErrorField]
```

An error definition is composed by a description and a short acronym. We also specify an error status, either a failure case or a success case.

```
data ErrorField = ErrorField ErrorStatus String String
```

```
data ErrorStatus = Failure
```

```
    | Success
```

```
deriving Eq
```

3.1.2 Embedding Fugu in Haskell

Instead of designing a parser, possibly using Parsec, we chose to embed the Fugu language inside Haskell. This means that we are going to define some Haskell functions, operating on the AST. By combining these functions, we will be able to build a complete AST that can be processed by Fugu's back-end.

First of all, we can build an *ErrorField* out of the error acronym and description, using one of these two combinators:

```
success, failure :: String → String → ErrorField
```

```
success acronym description = ErrorField Success acronym description
```

```
failure acronym description = ErrorField Failure acronym description
```

Given a list of such fields and a global name for them, we can then build an *ErrorClass*:

```
errors :: String → [ErrorField] → ErrorClass
```

```
errors className errors = ErrorClass className errors
```

And, finally, given a list of such classes, we get a valid error definition "file":

```
define :: [ErrorClass] → Errors
```

```
define errors = errors
```

3.1.3 Our Example in Fugu

If we translate our informal description of Section 2.2, this leads to the following code:

```
testE = define [
  errors "system_err" [
    success "SYS_OK" "No error",
    failure "SYS_FAIL" "Failed",
    success "SYS_ILIA" "Mafioso Pasta",
    failure "SYS_TEM" "Kernel Hacker",
    failure "SYS_IPHUS" "Analphabet Greek"
  ],
  errors "pci_err" [
    success "PCI_OK" "No PCI error",
    failure "PCI_CAP_NOK" "Lost my cap",
    success "PCI_GET_CAP" "That is my cap"
  ]
]
```

3.2 Filet-O-Fish Back-end

At this point, there is no way we can postpone the implementation of the compiler back-end. So, *Ave, Caesar, Morituri Te Salutant.*

3.2.1 Defining Type Aliases

We start with some type aliasing. In its basic form, FoF only knows *primitive* types such as integers, floats, and user-defined structures, unions, and arrays. Therefore, if we are to use the standard *bool* or *char*, we have to inform FoF of their existence as well as to which primitive type they match to.

```
boolT :: TypeExpr
boolT = typedef uint64T "bool"
charT :: TypeExpr
charT = typedef uint8T "char"
```

Similarly, we can define our own aliases:

```
err_codeT :: TypeExpr
err_codeT = typedef uint16T "err_code"
err_code :: Int → PureExpr
err_code = uint16 ∘ toInteger
errval_tT :: TypeExpr
errval_tT = typedef uint64T "errval_t"
```

3.2.2 Defining General Functions

Implementing the *err_no*, *err_push*, *err_is_ok*, and *err_is_fail* consists in a straightforward translation from C to FoF. Let us give a closer look at *err_no*, for instance.

The body of *err_no* consists in doing some bit twiddling on the error stack and returning this value:

```
err_no_int :: [PureExpr] → FoFCode Def
err_no_int (err : []) =
  returnc (err .&. ((uint64 1 . << . uint64 17) . - . uint64 1))
```

We build a function out of the function body thanks to the *def* operator. Along with the function body, we have to pass the return type as well as the parameters types:

```
err_noF :: (Def :<: f) => FoFCode f
err_noF = def [Static, Inline] "err_no" err_no_int err_codeT [errval_tT]
```

And we are done: *err_noF* corresponds to the C *err_no* function. We will see later on how it can be called.

Similarly, we define *err_pushF*. This time, we declare the function body in a **where** clause. Note, however, that we have to provide the type of *err_push_int* as it cannot be inferred by the Haskell compiler.

```
err_pushF :: (Def :<: f) => Semantics f PureExpr
err_pushF = def [Static, Inline] "err_push" err_push_int errval_tT [errval_tT, err_codeT]
  where err_push_int :: [PureExpr] -> Semantics Def PureExpr
        err_push_int (err : code : []) =
          returnc ((err . << . uint64 16) .|. code)
```

And the same goes for *err_is_okF* and *err_is_failF*:

```
err_is_okF :: (Def :<: f) => FoFCode f
err_is_okF = def [Static, Inline] "err_is_ok" err_is_ok_int boolT [errval_tT]
  where err_is_ok_int :: [PureExpr] -> Semantics Def PureExpr
        err_is_ok_int (err : []) =
          do
            err_no <- err_noF
            err_no_e <- call err_no [err]
            returnc (err_no_e .%. uint64 2)

err_is_failF :: (Def :<: f) => Semantics f PureExpr
err_is_failF = def [Static, Inline] "err_is_fail" err_is_fail_int boolT [errval_tT]
  where err_is_fail_int :: [PureExpr] -> Semantics Def PureExpr
        err_is_fail_int (err : []) =
          do
            err_no <- err_noF
            err_no_e <- call err_no [err]
            returnc (uint64 1 . - . (err_no_e .%. uint64 2))
```

The case of *err_assert_ok* and *err_assert_fail* is more interesting. Indeed, they respectively call *err_is_ok* and *err_is_fail*. The function call is performed by the operator *call*, provided with the right parameter(s). Note that we need to bind *err_is_okF* to a variable *err_is_ok* before using it. This is a classical monadic idiom.

```
err_assert_okF :: (Def :<: f, Assert :<: f) => Semantics f PureExpr
err_assert_okF = def [Static, Inline] "err_assert_ok" err_assert_ok_int voidT [errval_tT]
  where err_assert_ok_int :: [PureExpr] -> Semantics (Assert : + : Def) PureExpr
        err_assert_ok_int (err : []) =
          do
            err_is_ok <- err_is_okF
            err_no_e <- call err_is_ok [err]
            assert err_no_e

err_assert_failF :: (Def :<: f, Assert :<: f) => Semantics f PureExpr
err_assert_failF = def [Static, Inline] "err_assert_fail" err_assert_fail_int voidT [errval_tT]
  where err_assert_fail_int :: [PureExpr] -> Semantics (Assert : + : Def) PureExpr
        err_assert_fail_int (err : []) =
          do
```

```

err_is_fail ← err_is_failF
err_no_e ← call err_is_fail [err]
assert err_no_e

```

3.2.3 Defining Parameterized Functions

Our first example of parameterized functions is the set of *err_is_** functions, with one such function per error code. This example illustrates the flexibility of FoF for meta-programming.

We provide *err_is_list* with a list of pairs (error acronym, error code). For each element of this list, *err_is_list* defines a function whose name follows the acronym and that tests for equality with the error code. Hence, *err_is_list* returns a list of function definitions.

```

err_is_list :: (Enum <: f, Def <: f) ⇒ Enumeration → [String] → [FoFCode f]
err_is_list enum codes = map err_is codes
  where err_is name =
    def [Static, Inline] ("err_is_" ++ map toLower name) err_is_f boolT [errval_tT]
    where err_is_f :: [PureExpr] → Semantics (Def : + : Enum) PureExpr
      err_is_f (err : []) = do
        err_no ← err_noF
        err_no_e ← call err_no [err]
        code ← ofEnum enum name
        returnc (err_no_e . == . code)

```

A more involved example is *err_get_array*, which is used to build the *err_getdomain*, *err_getstring*, and *err_getcode* functions. Hence, *err_get_array* takes as first argument the postfix of the *err_get** target. It is also provided a reference to the array that contains the descriptive strings. And the last argument consists of a list of pairs (error code, array index).

The body of the *err_get_array* is a small generalization over our C examples: it calls *err_no* on the argument and switches over it. For each error code, it reads in the provided array, at the corresponding index. The default case simply consists in reading at the latest position in the array, where “undefined” or its variant is written.

```

err_get_array :: (Conditionals <: f, Def <: f, Array <: f) ⇒
  String → PureExpr → [(Int, PureExpr)] →
  PureExpr → Semantics f PureExpr
err_get_array name array codes defaultCode =
  def [Static, Inline] ("err_get" ++ name) err_getstring_int (ptrT charT) [errval_tT]
  where err_getstring_int :: [PureExpr] → Semantics (Def : + : Array : + : Conditionals) PureExpr
    err_getstring_int (err : []) =
      do
        err_no ← err_noF
        err_no_e ← call err_no [err]
        switch err_no_e
          [(uint64 $ toInteger code, (return_array array index
            :: Semantics (Def : + : Array) PureExpr))
           | (code, index) ← codes]
          (return_array array defaultCode
            :: Semantics (Def : + : Array) PureExpr)
        returnc (uint64 0)
    return_array array index =
      do

```

```

x ← readArray array index
returnc $ cast (ptrT charT) x

```

The last function is `err_print_calltrace`. Although longer, this function does not involve technical aspects of FoF. We can observe the use of `if` and `while` operators. As well as the definition and usage of *references*. We also make use of the `printf` operator.

```

err_print_calltraceF :: (Conditionals <: f, Def <: f, Ref <: f, Printf <: f) =>
  PureExpr → PureExpr → PureExpr → FoFCode f
err_print_calltraceF err_getdomain err_getstring err_getcode =
  def [Static, Inline] "err_print_calltrace" err_print_calltrace_int voidT [errval_tT]
  where err_print_calltrace_int :: [PureExpr] → Semantics (Ref : + :
    Def : + :
    Conditionals : + :
    Printf) PureExpr
    err_print_calltrace_int (err : []) =
      do
        err_no ← err_noF
        err_is_fail ← err_is_failF
        err_ref ← newRef err
        ifc (do
          is_fail_e ← call err_is_fail [err]
          return is_fail_e
          :: Semantics (Ref : + : Def) PureExpr)
        (do
          x ← newRef $ uint16 0
          while (do
            err_ref_val ← readRef err_ref
            err_ref_no ← call err_no [err_ref_val]
            writeRef x err_ref_no
            return $ err_ref_no .! = . (uint16 0)
            :: Semantics (Ref : + : Def) PureExpr)
          (do
            err_ref_val ← readRef err_ref
            err_ref_dom ← call err_getdomain [err_ref_val]
            err_ref_str ← call err_getstring [err_ref_val]
            err_ref_code ← call err_getcode [err_ref_val]
            printf "Failure: (%15s) %20s [%10s]\n"
              [err_ref_dom, err_ref_str, err_ref_code]
            writeRef err_ref (err_ref_val . >> . (uint64 16))
            :: Semantics (Ref : + : Def : + : Printf) PureExpr)
          :: Semantics (Ref : + : Def : + : Conditionals : + : Printf) PureExpr)
        (do return void
          :: Semantics Def PureExpr)

```

3.2.4 Defining Data-Structures

The following codes involves more Haskell-specific data-processing of the AST, and less FoF-specific manipulations. However, this quasi-absence of FoF is also a good sign: FoF succeeds in being non-intrusive while being able to provide strong guarantees on the compiler's correctness.

Compiling the Labels The first step here is to extract the labels, ie. the error codes and descriptions, from the Error Definition. Once extracted, we turn them into FoF strings, thanks to *makeStrings*.

```
err_strings :: Errors → ([[PureExpr]], [[PureExpr]], [[PureExpr]])
err_strings errors =
  (makeStrings domains "Undefined",
   makeStrings errcodes "UNDEFINED",
   makeStrings errdescs "Undefined")
  where domains = [domain | ErrorClass domain _ ← errors]
        (errcodes, errdescs) = unzip $ concat [[(code, descr)
          | ErrorField _ code descr ← fields]
        | ErrorClass _ fields ← errors]
```

And, actually, a FoF string is simply a static array of characters:

```
makeStrings :: [String] → String → [[PureExpr]]
makeStrings domains undefinedC = map err_domain_string (domains ++ [undefinedC])
  where err_domain_string domain = map charc domain ++ [charc '\0']
```

Using *err_strings*, the following function builds the 3 static arrays *err_domains*, *err_codes*, and *err_msgs*. Remember that *err_strings* returns a triple of lists of strings. We instantiate each of these strings as static arrays, hence obtaining a list of arrays. Finally, we instantiate these list of arrays as static arrays. Hence, we obtain 3 static arrays, each containing some static arrays: the various strings.

```
newStaticArrayIndex :: (Array <: f) ⇒
  String →
  (Int, [PureExpr]) →
  Semantics f PureExpr
newStaticArrayIndex name (i, x) = newStaticArrayN (name ++ show i) x
```

```
err_arrays :: (Array <: f) ⇒ Errors → Semantics f (PureExpr, PureExpr, PureExpr)
err_arrays errors =
  do
    let (domains, errcodes, errdescs) = err_strings errors
        errdomains_str ← sequenceSem $ map (newStaticArrayIndex "errdomains_") $ zip [1..] domains
        errcodes_str ← sequenceSem $ map (newStaticArrayIndex "errcodes_") $ zip [1..] errcodes
        errdescs_str ← sequenceSem $ map (newStaticArrayIndex "errdescs_") $ zip [1..] errdescs
        errdomains_arr ← newStaticArrayN "err_domains" errdomains_str
        errcodes_arr ← newStaticArrayN "errcodes" errcodes_str
        errdescs_arr ← newStaticArrayN "errdescs" errdescs_str
    return (errdomains_arr, errcodes_arr, errdescs_arr)
```

At this point, we have compiled the labels.

Compiling the Enumeration Before compiling the enumeration, we need to compute two lists of random, unique, odd and even numbers. This is achieved by the following function, which implementation is hidden for the sake of brevity. It takes a random number generator, the number *n* of desired integers, and returns a pair composed by *n* even numbers and *n* odd numbers.

```
mkRandomUnique :: StdGen → Int → ([Int], [Int])
```

Using these lists of even and odd numbers, we assign an identifier to each error and return this mapping as a list of triples, containing the domain, the acronym, and the unique identifier. Depending whether the error indicates a success or a failure, we pick the identifier, respectively, among the odd or even numbers.

```

mkEnum :: Errors → [Int] → [Int] → [(String, String, Int)]
mkEnum errors evenNumbers oddNumbers = reverse fields
  where (fields, -, _) = foldl mkEnumField ([], evenNumbers, oddNumbers) errorTypes
        errorTypes = concat [(typ, name, dom)
                              | ErrorField typ name _ ← fields]
                              | ErrorClass dom fields ← errors]
mkEnumField (acc, evenNumbers, oddNumbers) (typ, name, dom)
  | typ ≡ Failure =
    ((dom, name, head evenNumbers) : acc,
     tail evenNumbers,
     oddNumbers)
  | typ ≡ Success =
    ((dom, name, head oddNumbers) : acc,
     evenNumbers,
     tail oddNumbers)

```

3.2.5 Putting Things Together

The back-end now consists in a straightforward recollection of the various, previously developed pieces. Its type is the following:

```

backendHeader :: StdGen → Errors →
  FoFCode (EnumE : + :
    Assert : + :
    Ref : + :
    Def : + :
    Array : + :
    Conditionals : + :
    Typedef : + :
    Printf)

```

Meaning that it takes a random number generation, an Error AST and it generates a Semantics. We can notice which components of FoF are used here: Enumerations, Assert, References, Functions, Static arrays, Conditionals, Type definitions, and Printf.

The back-end executes the following actions:

```

backendHeader gen errors =
  do

```

Compute the labels arrays:

```

  (err_domains,
   err_codes,
   err_descs) ← err_arrays errors

```

Compile the type-definitions:

```

  alias errval_tT
  alias err_codeT
  aliasE "<stdlib.h>" charT
  aliasE "<stdbool.h>" boolT

```

Compile the enumeration

```
newEnum "err_code" enumErrCodes
```

Compile the functions:

```
err_no ← err_noF  
err_push ← err_pushF  
err_is_ok ← err_is_okF  
err_is_fail ← err_is_failF  
err_assert_ok ← err_assert_okF  
err_assert_fail ← err_assert_failF
```

Compile the parameterized functions:

```
err_is_tests ← sequenceSem $ err_is_list enumErrCodes errorCodes
```

Compile the array accessors:

```
err_getstring ← err_get_array "string" err_descs mapIdDescription (err_code noCodes)  
err_getcode ← err_get_array "code" err_codes mapIdAcronym (err_code noCodes)  
err_getdomain ← err_get_array "domain" err_domains mapIdDomain (err_code noDomains)
```

Finally, compile the call-trace printer:

```
err_print_calltrace ← err_print_calltraceF err_getdomain err_getstring err_getcode
```

And we are done.

```
return void
```

3.2.6 Compiling the Test-Case

At this stage, we just have to open a file, call FoF's *compile* function on the back-end applied to the test-case, and we are done!

```
Main :  
main :: IO ()  
main = do  
  let gen = mkStdGen 1  
  fileC ← openFile "test_error.h" WriteMode  
  hPutStr fileC $ show $ fst $ compile (backendHeader gen testE) emptyBinding  
  hClose fileC
```

Thanks for your attention!