Effective CAL

Bo Ilic

Last modified: August 24, 2007

Introduction

What follows is a series of essays describing particular topics of interest in writing readable, maintainable, extensible and efficient CAL programs. They are intended to be in the spirit of the Effective C++ and Effective Java books- intermediate in level, but useful general information for pragmatic programming. These essays assume that you are familiar with the basics of CAL such as could be obtained by reading the CAL User's Guide, learning to use the Interactive CAL Environment (ICE) and the GemCutter, reading through some of the code in the standard CAL modules, and writing a bit of CAL for yourself. The essays are not self-contained, so don't worry if you don't understand everything at first. Hopefully things will become clearer as you read on. Please send feedback, including suggestions for new "Effective CAL" essays to the CAL Language Discussion forum on Google Groups (http://groups.google.com/group/cal_language).

1. What is Weak Head Normal Form and why is it useful?

CAL is a lazy language, so by default, when asked to evaluate an expression it does as little work as possible before stopping the evaluation and allowing the client to do something else. This smallest unit of work is called *evaluating the expression to weak head normal form* (or *WHNF* for short). Understanding WHNF is important for understanding the operational semantics of CAL, that is, the actual steps that CAL takes in evaluating an expression. For example, to evaluate a case expression,

```
case conditionExpr of …
```

the first thing that CAL will do is evaluate conditionExpr to WHNF. At that point it will be able to choose which branch of the case to continue evaluating.

What WHNF means depends on the particular form of the expression.

Literal values and literal functions are in WHNF. For example, the following expressions are in WHNF:

```
2.0
'm'
"apple"
Math.cos
List.head
Prelude.equals
```

Values of a foreign type that are actual Java objects of the corresponding implementation type are in WHNF. For example, the CAL type JList corresponds to the Java type

java.util.List. If a JList value is an actual java.util.List object, as opposed to a computation that results in a java.util.List object, then that value is in WHNF.

Any time a function or class method has been supplied with enough arguments to be evaluated, then that expression is *not* in WHNF. For example, the following expressions are listed in pairs, the first is not in WHNF, the second is the first after having been evaluated to WHNF:

```
id 99.0
WHNF 99.0

2.0 + 5.0
WHNF 7.0

(2.0 + 1.0) * 5.0
WHNF 15.0
```

List.outputList [3 :: Int, 1, 4]
WHNF a java.util.List object holding 3 java.lang.Integer values 3, 1 and 4.

```
"help" ++ "ful"
WHNF "helpful"

Math.cos 0
WHNF 1.0
```

Note that +, * and ++ are the operator forms of the class methods Prelude.add, Prelude.multiply and Prelude.append.

It is important to note that the above rule does not apply to *data constructors* that have been supplied with enough arguments to be evaluated. In fact, this is one of the major differences between data constructors and functions or class methods. Luckily, data constructor names always start with a capital letter while function or class method names start with a lower case letter so it is easy to distinguish these cases.

A data constructor is always in WHNF, regardless of the form of its arguments. For example the following are examples of expressions in WHNF.

```
Left ("aard" ++ "vark")
Just (2.0 + 9.0)
[sin 0, cos 0, 3 + 5]
[]
GT
True
```

Note, in the case of the list
`[sin 0, cos 0, 3 + 5]`

This is shorthand notation for
```
(sin 0) : (cos 0) : (3 + 5) : []
```

which using the textual form for the : and [] operators (Cons and Nil) is just
```
Cons (sin 0) (Cons (cos 0) (Cons (3 + 5) Nil))
```

This is just a data constructor applied to 2 arguments, so it is by definition in WHNF.

Record values and tuple values (which are just a special kind of record value) are automatically in WHNF. For example, the following expressions are in WHNF:

```
(2.0, 'm', "pear")

(2.0 + 5.0, "this" ++ "that")

{name = "Anton", age = 1.0 + 1.0, occupation =
String.fromList ['b', 'a', 'b', 'y']}
```

Expressions which are function calls for which not enough arguments have been supplied are in WHNF. For example, the following expressions are in WHNF

```
//power takes 2 arguments, only 1 supplied
power (2.0 + 3.0)
//add takes 2 arguments, only 1 supplied
add (sin 0)
```

Expressions which are lambda expressions for which not enough arguments have been supplied are in WHNF. For example the following are in WHNF:
```
(\x y -> x + y) 2.0
(\x -> head [x])
(\x y -> cos 0 + x - y) 100.0
```

let expressions are in WHNF if the expression following the "in" is in WHNF.

Expressions in forms other than mentioned above are not in WHNF. In particular, the following are not in WHNF

applications, other than those forms explicitly mentioned above
lambda expressions with enough arguments supplied
if-then-else expressions
case expressions
data constructor field selection expressions
record field selection expressions
record extension expressions

The following expressions are listed in pairs. The first is not in WHNF and the second is the evaluation to WHNF:

```
fromJust (Just (2.0 + 9.0))
WHNF 11.0

(2.0 + 5.0, "this" ++ "that").#2
WHNF "thisthat"

tail [sin 0, cos 0, 3 + 5]
WHNF [cos 0, 3 + 5]

head [add (2.0 + 3.0)]
WHNF 5.0

(\x y -> x + y) 3.0 7.0
WHNF 10.0

(\r -> {r | name = "Anton"}) {occupation = "baby", age =
1.0 + 1.0}
WHNF {name = "Anton", occupation = "baby", age = 1.0 + 1.0}

if True then Just (cos 0) else Nothing
WHNF Just (cos 0)

case compare 10.0 20.0 of LT -> Left (2.0 + 5.0); _ ->
Right (6.0 + 8.0);
WHNF Left (2.0 + 5.0)

case compare 10.0 20.0 of LT -> 2.0 + 5.0; _ -> 6.0 * 8.0;
WHNF 7.0

(Just (Left ("abc" ++ "def"))).Just.value
WHNF Left ("abc" ++ "def")
```

2. How are expressions in CAL evaluated?

To evaluate an expression in CAL means to evaluate an expression to weak-head normal form or WHNF. CAL will always stop at that point, and allow clients, such as an enclosing context, to determine what to do next.

This may seem surprising because of the behavior of ICE.

For example, if on the ICE command line you type the following expression, which is already in WHNF:

```
Just (2.0 + 3.0)
```

ICE returns
```
(Prelude.Just 7.0)
```

The reason for this is that expressions on the ICE command line are automatically prefixed by Prelude.output to convert them to a Java object. What is really being run is
```
Prelude.output (Just (2.0 + 3.0)) :: JObject
```

The behavior of the output class method on the Maybe Double type is defined by the Outputable Maybe and Outputable Double instances. In this case they will output to a java value of type org.openquark.cal.foreignsupport.module.Prelude.MaybeValue. This Java object then has the toString() Java method called on it which produces the output above.

To evaluate a case expression, "case conditionExpr of …", the conditionExpr is evaluated to WHNF. conditionExpr will then be either a data constructor application, a literal Int, a literal Char, or a record value. At this point, CAL can decide which branch of the case expression to follow, and it evaluates that branch to WHNF. Note that even if the case expression has only 1 branch, the conditionExpr will first be evaluated to WHNF.

```
case (Just (2.0 + 7.0)) of Nothing -> "bad"; Just x ->
"good";
```

returns "good", without ever evaluating 2.0 + 7.0, since "Just (2.0 + 7.0)" is already in WHNF.

How can we tell that 2.0 + 7.0 is not evaluated? The main techniques here involves using the trace, error and seq functions to modify the expression such that a failure of some sort (either a message to the console when using trace, or terminating execution using error) is triggered if the offending expression is evaluated. For example, the following expressions typed into the ICE console also just return "good", without logging a message or an exception:

```
case (Just (trace "we have a problem" (2.0 + 7.0))) of
Nothing -> "bad"; Just x -> "good";

case (Just (error "we have a problem" `seq` (2.0 + 7.0)))
of Nothing -> "bad"; Just x -> "good";
```

You can alter the original expression in a variety of ways. These still return "good":

```
case (Just (error "we have a problem")) of Nothing ->
"bad"; Just x -> "good";

case (Just ((error "we have a problem") + 7.0)) of Nothing
-> "bad"; Just x -> "good";
```

As mentioned above, even if the case expression has only one branch, the conditionExpr is still evaluated. You can see this in ICE for example with the expression:
case (error "will stop here") of {name, age} -> "hello";

This will terminate in an error displaying the message "will stop here".

Evaluating an if-then-else expression is similar to evaluating a case-expression. The conditionExpr is evaluated to WHNF. Since the conditionExpr in an if-then-else has type Boolean, this must be either the value True or False. Then the appropriate branch is selected, and evaluated to WHNF.

In the functional programming literature, the evaluation algorithm of CAL is said to evaluate expressions in *leftmost-outermost* order. What this means is that, first of all, outermost expressions are evaluated prior to inner expressions being evaluated. For example, in expressions such as "f expr1 (g expr2)", the application of f to the 2 arguments expr1 and (g expr2) is performed prior to the application of g to its argument expr2. The leftmost order means that for a function application (f expr1 expr2), expr1 is evaluated prior to expr2 i.e. in left-to-right order. This is true for many functions, for example, all foreign functions evaluate their arguments to WHNF in left-to-right order. However, it is not true in general of all CAL functions, but rather it is a property of the individual function f.

For example, the function
```
f x y = "hello";
```
will not evaluate either of its arguments to WHNF, since it ignores its arguments.

The function
```
f :: Int -> Int -> Int;
f x y = y - x;
```
will evaluate y to WHNF first, followed by evaluating x to WHNF.

The function
```
iff :: Boolean -> a -> a -> a;
iff b x y = if b then x else y;
```

will evaluate b to WHNF first, followed by evaluating x to WHNF if b is TRUE and y to WHNF if b is false.

So for example, both expressions below return "OK".
```
iff True "OK" (error "we have a problem")
iff False (error "we have a problem") "OK"
```

Evaluating an application expression first evaluates the left-hand-side of the application to WHNF, and then attempts to apply the resulting function. For example, the application
`expr1 expr2 expr3`
with parentheses inserted to account for the left ascociativity of application is:
`(expr1 expr2) expr3`

So to evaluate this, we first must evaluate (expr1 epxpr2) to WHNF, which means first evaluating expr1 to WHNF.

For example,
```
(head [power]) (1.0 + 1.0) (2.0 + 1.0)
```
will first evaluate `(head [power])` to WHNF getting
```
power
```

It will then evaluate "power (1.0 + 1.0)" to WHNF. This is already in WHNF since power is a function that takes 2 arguments, and only 1 is supplied.
So it now can apply the function power on its 2 arguments, power (1.0 + 1.0) (2.0 + 1.0).
Since power is a foreign function, the arguments are evaluated to WHNF in left-to-right order:
```
power 2.0 (2.0 + 1.0)
power 2.0 3.0
8.0
```

Here is another example of reducing applications:
```
(head [fst]) (cos, sin) (5.0 - 5.0)
//fst is in WHNF, but fst (cos, sin) is not
//so we reduce this first.
fst (cos, sin) (5.0 - 5.0)
cos (5.0 - 5.0)
cos 0.0
1.0
```

3. What is the seq function and what is it good for?

The seq function is a useful function for controlling the order in which reduction to WHNF occurs i.e. sequencing reductions. When an expression "seq expr1 expr2" is evaluated to WHNF, seq will first evaluate expr1 to WHNF, and then evaluate expr2 to WHNF, and then return the evaluated expr2.

For example, running
```
(error "will call") `seq` "apple"
```
in ICE results in an error with "will call" displayed on the console.

Running
```
(Debug.sleep 5000 "sad") `seq` ("hap" ++ "py")
```
will result in a delay of 5 seconds, followed by the console displaying the returned value of "happy".

It is important to note that seq expressions are not guaranteed to be evaluated. It forces *sequencing* of evaluations. For example,
```
snd (error "will not call" `seq` "apple", "pear")
```
returns "pear" while

```
fst (error "will call" `seq` "apple", "pear")
```
terminates in an error, displaying "will call" on the console.

In the fist example, since the expression
```
error "will not call" `seq` "apple"
```
was never evaluated to WHNF, the first argument of seq never needed to be evaluated to WHNF, and so the error never happened.

seq is used mainly for two different purposes.
a) writing code that makes use of side-effects, in which the order of execution of operations is important.
b) controlling space (memory) usage.

4. What does strictness mean?

First I'll give the official definition, and then show what it really means by examples. A function "f x1 x2 … xn = expression;" of n arguments is said to be *strict in its jth argument xj* if for all possible choices of expressions e1, e2, … en, reducing "f e1 e2 … en " to WHNF will always result in either
a) reducing ej to WHNF
b) not succeeding in reducing "f e1 e2 … en" to WHNF because the evaluation hangs or terminates in an error.

In other words, when CAL evaluates any fully saturated application of f, it will end up evaluating the jth argument, or die trying. A function f not strict in its argument xj is said to be *lazy in xj*.

It is important to note that strictness is a more inclusive notion that plinging i.e. if the jth argument of f is plinged, f is strict in xj, but the converse is not true. I'll explain more about plinging in another point.

Here are some examples:
```
tuple2 :: a -> b -> (a, b);
tuple2 x y = (x, y);
```
tuple2 is not strict in any of its arguments x, y

```
list3 :: a -> a -> a -> [a];
list3 x y z = [x, y, z];
```
list3 is not strict in any of its arguments x, y, z.

For example, to see non-strictness of list3 we can type in ICE:
```
list3 (error "x") (error "y") (error "z") `seq` "OK"
```
This will return "OK".

By the definition of the seq function, the expression
```
list3 (error "x") (error "y") (error "z")
```

was evaluated to WHNF. Since it evaluating the whole expression returns "OK" successfully, we know that condition b) above does not hold. If any of the arguments of list3 were evaluated to WHNF, the corresponding error function would be called. Since we didn't get an error exception, condition a) did not occur. Thus list3 is not strict in any of its arguments.

Intuitively, list3 constructs a 3-element list. Until the actual elements of the list are examined, the arguments x, y and z do not need to be evaluated. For example, client code that simply wanted to compute the length of the list would never need to evaluate the arguments. That is why

```
List.length (list3 (error "x") (error "y") (error "z"))
```

returns 3.

Foreign functions are strict in all of their arguments. Indeed, if f is a foreign function, then the expression

```
f e1 … en
```

is evaluated by evaluating e1, e2, …, en each to WHNF in left-to-right order to obtain Java object values, and then calling the Java function corresponding to f with the Java argument values. The only way this can fail is if exceptions or non-termination (hanging) happens.

Functions are commonly lazy in some arguments and strict in some arguments. For example, the function fromMaybe is strict in the maybeValue argument but lazy in the defaultValue argument.

```
fromMaybe :: a -> Maybe a -> a;
public fromMaybe defaultValue maybeValue =
    case maybeValue of
    Just value -> value;
    Nothing -> defaultValue;
    ;
```

To see this, note that:

```
fromMaybe (error "defaultValue evaluated") (Just 2.0)
```

returns 2.0. Thus, fromMaybe is lazy in the defaultValue argument.

The defining expression of fromMaybe starts with a "case maybeValue of …". Evaluating a case expression to WHNF first evaluates the conditionExpr to WHNF. In this case this means maybeValue is evaluated to WHNF. Thus, fromMaybe is strict in the maybeValue argument.

As a side note, the actual definition of Prelude.fromMaybe plings the maybeValue argument. This does not affect the strictness of the maybeValue argument- it is still strict even without this plinging.

Thinking about what a function does is a good way to guess at what its strictness is likely to be. For example, fromMaybe defaultValue maybeValue, checks if maybeValue is a Just value, and if so returns the underlying value, and otherwise returns the defaultValue.

Clearly to see if maybeValue is a Just or a Nothing, maybeValue must always be evaluated to WHNF. Also, defaultValue is only ever needed in the Nothing case, so there will be situations when it is not evaluated i.e. fromMaybe should be lazy in defaultValue. This approach to deducing strictness is not rigorous, but it can be helpful when designing a new function, or creating unit tests. In general, all things being equal, it is a good thing for a function to be lazy in an argument, if this is possible. This is because not evaluating an argument is a potential efficiency gain for clients when using the function.

5. What is plinging? (part1 – plinging of function arguments)

Plinging can be done in 2 contexts: for algebraic function definitions and for algebraic data declarations.

For an algebraic function definition, such as

```
f x !y z !w !t = expression;
```

in which the y, w and t arguments are plinged, this has exactly the same operational meaning as

```
f x y z w t = y `seq` w `seq` t `seq` expression;
```

or alternatively,

```
f x y z w = seq (seq (seq y w) t) expression;
```

/Notice that only the plinged arguments in the original are sequenced. x and z are not sequenced. Also, the sequencing is in argument order y, w and t.

In other words, when evaluating a fully saturated application of f to WHNF, the plinged arguments are evaluated to WHNF, in the order in which they are plinged, prior to the defining expression itself being evaluated to WHNF.

In particular, if a function has a plinged argument, then that argument is automatically a strict argument of the function.

Plinging can alter the lazy reduction order of a function. For example, the functions sub1 and sub2 are both strict in their arguments x and y:

```
sub1 x y = y - x :: Double;
sub2 !x y = y - x :: Double;
```

Evaluating the expression:

```
sub1 (error "arg x") (error "arg y")
```

will terminate in an error and display "arg y".

However, evaluating the expression:

```
sub2 (error "arg x") (error "arg y")
```

will terminate in an error and display "arg x".

Here are 4 separate functions for making pairs. They differ in their strictness due only to plinging.

```
//lazy in x and y
pair x y = (x, y);

//strict in x and y
pairStrict !x !y = (x, y);

//strict in x, lazy in y
pairFstStrict !x y = (x, y);

//lazy in x, strict in y
pairSndStrict x !y = (x, y);
```

So for example,

```
(pair 2.0 (error "arg y evaluated")).#1
```
returns 2.0

```
(pairSndStrict 2.0 (error "arg y evaluated")).#1
```
terminates in an error and prints "arg y evaluated" to the console.

Note that plinging can be applied to any function argument. This includes local functions and lambda expressions e.g. `\!x !y -> x*x + y*y`.

6. What is plinging? (part2 – plinging of data constructor arguments)

Another place where plinging is used is in algebraic type definitions. This is a more fundamental concept than plinging of function arguments because plinging of data constructor arguments, unlike plinging of function arguments, can not be inferred by the compiler- it expresses something about the design intent of the type. For example, Haskell supports plinging of data constructor arguments, but not plinging of function arguments (although Haskell indirectly supports the semantic effects of plinging of function arguments via the seq function). Plinging of data constructor arguments is a useful technique for controlling both the space usage of programs during evaluation, as well as the in-memory footprint of already evaluated values.

Each field of each data constructor within a type can be either plinged or unplinged. For example, in LazyPair, none of the fields are plinged, in StrictPair both fields are plinged, and in FirstStrictPair only the fst field is plinged.

```
data LazyPair a b =
    LP
        fst :: a
```

```
        snd :: b
    ;

data StrictPair a b =
    SP
        fst :: !a
        snd :: !b
    ;

data FirstStrictPair a b =
    FSP
        fst :: !a
        snd :: b
    ;
```

Plinging of the fields of a data constructor affects how that data constructor is reduced to WHNF. In particular, when evaluating a data constructor to WHNF, the plinged fields will first themselves be evaluated to WHNF, in the order in which they are declared in the data constructor's definition.

For example, evaluating

```
(LP 7.0 (error "pear")).LP.fst
```

returns the value 7.0 while evaluating

```
(SP 7.0 (error "pear")).SP.fst
```

terminates in an error and displays "pear" on the console. This is because in the LazyPair case, evaluating the data constructor field selection first causes

```
LP 7.0 (error "pear")
```

to be evaluated to WHNF. It is already in WHNF, and so evaluation can proceed with returning the value of the fst field. However, in the StrictPair case, evaluating

```
SP 7.0 (error "pear")
```

to WHNF first involves evaluating `7.0` to WHNF and then `(error "pear")` to WHNF. At this point evaluation terminates with the error call.

Note that if the SP expression above is itself never evaluated to WHNF, then the error call will not occur. For example, the following expression returns 3.0:

```
(3.0, SP 7.0 (error "pear")).#1
```

One consequence is that when a plinged data constructor is in WHNF, then its plinged fields are also in WHNF. This has the semantic effect that types with plinged fields have fewer values than a similar type with unplinged fields. Plinging data constructor fields is

a useful technique for controlling the space usage of a program. For example, it is impossible for an evaluated value of `type StrictPair Int Int` to hold onto a bulky unevaluated computation in one of its fields, since they must be evaluated to WHNF, which in the case of Int means a simple Int literal value. A similar effect can be had with LazyPair by always constructing its values using a helper function such as lazyPair:

```
lazyPair !x !y = LazyPair x y;
```

Since the x and y arguments of lazyPair are plinged, this will do the trick.

The LazyPair type is very similar to the standard CAL tuple type (a, b). Tuple and record types in CAL always have lazy fields. A similar technique of controlling space usage when constructing CAL tuples is to use a function such as pairStrict, discussed in the previous essay:

```
pairStrict !x !y = (x, y);
```

When a field is plinged, and the declared type of the field is a foreign type (or an enumeration type, which is internally represented as an int), then the storage requirements for evaluated values of the field are much less than if the field were not plinged.

For example, an evaluated StrictWidget internally holds onto 4 field values with Java types: java.lang.String, int, double, java.lang.Object. An evaluated LazyWidget holds onto 4 field values all of Java type org.openquark.cal.machine.CalValue, which is the Java implementation type of the CAL type Prelude.CalValue. If the field values of a LazyWidget value are eventually reduced to WHNF then there will be one level of indirection to values of the underlying Java types mentioned above.

```
data StrictWidget =
    SWidget
        name :: !String
        id :: !Int
        price :: !Double
        other :: !JObject
    ;


data LazyWidget =
    LWidget
        name :: String
        id :: Int
        price :: Double
        other :: JObject
    ;
```

I mentioned above how plinging can affect the set of values that a type can represent. This is especially pronounced in the case of recursively defined algebraic types. For example, shown below are four different list types, varying in their strictness constraints:

```
data LazyList a =
    LNil |
```

```
    LCons
        head :: a
        tail :: (LazyList a)
    deriving Eq, Ord;

data HeadStrictList a =
    HSNil |
    HSCons
        head :: !a
        tail :: (HeadStrictList a)
    deriving Eq, Ord;

data TailStrictList a =
    TSNil |
    TSCons
        head :: a
        tail :: !(TailStrictList a)
    deriving Eq, Ord;

data StrictList a =
    SNil |
    SCons
        head :: !a
        tail :: !(StrictList a)
    deriving Eq, Ord;
```

LazyList is essentially identical to CAL's standard list type, Prelude.List. LazyList and Prelude.List perform no evaluation unless needed by a client, and can be used to represent infinite data structures, such as a stream of generated outputs with no preset termination. However, they can be more inefficient if sooner or later all the list elements are evaluated.

StrictList is similar (but not identical to) CAL's Array type. The similarity lies in that whenever an Array, or StrictList is evaluated to WHNF, then *all* of its elements are first evaluated to WHNF. In particular, a StrictList can not be used to represent an infinite list in a useful way i.e. such that reduction to WHNF succeeds.

For example, here are the definitions of 4 infinite lists alternating the elements 1.0 and (error "apple"):

```
lazyList1 = LCons 1.0 (LCons (error "apple") lazyList1);

headStrictList1 = HSCons 1.0 (HSCons (error "apple")
headStrictList1);

tailStrictList1 = TSCons 1.0 (TSCons (error "apple")
tailStrictList1);

strictList1 = SCons 1.0 (SCons (error "apple")
strictList1);
```

Evaluating an expression to extract the first element of lazyList1 or headStrictList1:
`lazyList1.LCons.head`
`headStrictList1.HSCons.head`

both return 1.0.

Evaluating the third element of lazyList1:
`lazyList1.LCons.tail.LCons.tail.LCons.head`
also returns 1.0.

However, evaluating the third element of headStrictList1:
`headStrictList1.HSCons.tail.HSCons.tail.HSCons.head`

terminates in an error, displaying "apple" on the console. This is because during the list traversal, when unpacking the second HSCons node, the head field is evaluated, resulting in the call to error since the second element is (`error "apple"`).

Evaluating an expression to extract the first element of strictList1:
`strictList1.TSCons.head`

terminates in an error at the reduction point when the second element value is evaluated, displaying "apple" on the console.

However, evaluating an expression to extract the first element of tailStrictList1
`tailStrictList1.TSCons.head`
does not terminate in an error, since elements of the list itself are never evaluated. However, it does hang (eventually resulting in a java.lang.StackOverflowError). This is because the infinite list must be fully expanded out (without its values being evaluated) before the head value can be returned.

7. What is the accumulating parameters coding pattern, and why do I need it?

You may have noticed that many functions in CAL's standard libraries are implemented by calling local helper functions with more arguments. These extra arguments are called *accumulating parameters* and are used to store partial results required for the computation.

For example, the definition of the length function is (essentially):

```
length !list =
    let
        lengthHelper :: Int -> [a] -> Int;
        lengthHelper !acc !list =
            case list of
            [] -> acc;
            listHead : listTail ->
                lengthHelper (acc + 1) listTail;
            ;
```

```
    in
        lengthHelper 0 list;
```

The *acc* argument of lengthHelper represents the number of elements of the original list traversed so far while the *list* argument represents the remaining elements of the list yet to be visited. In a sense, these arguments can be considered as the partial results of the computation of the length of the original list. Since the acc argument is plinged, it will be a literal Int value at each reduction of lengthHelper. The standard definition of length is O(1) in space and O(n) in time. It is a key fact here that the accumulating parameter acc get evaluated to WHNF at each iteration. Sometimes this is accomplished by plinging, and sometimes through other means, such as the function naturally being strict in a given argument.

As an aside, the recursive call to lengthHelper is *tail-recursive*. From a performance point of view this is a good thing in that the lecc runtime will not use any heap in making the recursive call. At the java byte code level it is implemented as a goto. However, even without the tail-recursion optimization, length will be O(1) in space and O(n) in time, so the existence of the tail-recursion optimization is not a key point in understanding why length is implemented as it is. However, the tail-recursion optimization is an important topic in its own right and will be covered in another Effective CAL essay. A point to note now though is that it does not fundamentally affect asymptotic performance characteristics of your algorithm, but only the proportionality constant. (It can have a rather dramatic improvement on the proportionality constant though, since a goto instruction is much faster than allocating memory on the heap).

An alternative definition of length that does not use an accumulating parameters helper would be:

```
//warning- this is an example of how not to write length
lengthAlt :: [a] -> Int;
lengthAlt !list =
   case list of
   [] -> 0;
   listHead : listTail ->
       1 + lengthAlt listTail;
   ;
```

Although this definition is O(n) in time, it is also O(n) in space. Moreover, it is also O(n) in stack! To verify this, running an expression such as:

lengthAlt (upFromTo (1::Int) 10000)

on the ICE console will result in a java.lang.StackOverflowError being reported.

The first hint that a problem will occur is that reducing an application of lengthAlt to WHNF results in a literal Int. However, to return a literal Int from the recursion step of `1 + lengthAlt listTail`

requires that the second argument of the + operator, namely `lengthAlt listTail` be itself evaluated to WHNF. This requires a recursive call on the stack to obtain the result before the addition can be done. Eventually he function application will reduce to an intermediate computation of (1 + (1 + (1 + (…..)))) which when evaluated will run out of stack.

The solution in this particular case is to do the additions of 1 in an incremental fashion, one list element traversal at a time, rather than wait until the list is entirely traversed.

A general principle here is that it is bad to write directly recursive functions that perform computations in their returned value expression.

Listed below is essentially the definition of the List.reverse function. It uses a local recursive helper function reverseHelper, which has accumulating parameters source and result. At each recursive call, an element of source is moved to result. So result can be considered to be the partially reversed list, and source can be considered to be the part of the original list that still needs to be reversed.

```
reverse :: [a] -> [a];
reverse !list =
    let
        reverseHelper :: [a] -> [a] -> [a];
        reverseHelper !source result =
            case source of
            [] ->
                result;
            headSource : tailSource ->
                reverseHelper
                    tailSource (headSource : result);
            ;
    in
        reverseHelper list [];
```

reverse is an interesting contrast to the previous example of length, since its result type, [a] is a lazy type. However, the reverse function itself will return a list whose spine is fully evaluated, even though the elements themselves are not evaluated. It is O(n) in time and O(n) in space, where n is the length of the list to reverse. If the argument list is an infinite list, reverse not terminate.

Here is a more direct definition of reverse that does not use accumulating parameters.

```
//warning- this is an example of how not to write reverse
reverseAlt :: [a] -> [a];
reverseAlt !list =
   case list of
   [] -> [];
   listHead : listTail ->
       reverseAlt listTail ++ [listHead];
   ;
```

This definition is still O(n) in space, but it is O(n^2) in time. To see this, note that the reduction to WHNF of
`reverseAlt [1 :: Int, 2, 3, 4, 5]`

eventually reaches the step:
`(((([] ++ [5]) ++ [4]) ++ [3]) ++ [2]) ++ [1]`

Now, ++, the append class method on List, is O(n) in time, where n is the length of its first argument list. It doesn't take any time at all handling its second argument list. This is because ++ only needs to copy (the spine) of its first argument list, and not the second argument list- it can just reuse that value. (For details see the implementation of Prelude.appendList).

Thus, we can see the reason for the O(n^2) time behavior of reverseAlt- it is applying the append operator in a left associative way, when the work is being done in the first argument. This is precisely the wrong way to do it! As an aside, this is why the ++ operator is right-associate so that an expression like:

```
[] ++ [5] ++ [4] ++ [3] ++ [2] ++ [1]
```

in fact parses as:

```
[] ++ ([5] ++ ([4] ++ ([3] ++ ([2] ++ [1]))))
```

and thus can be evaluated without excessive copying of intermediate lists.

Note that the accumulating parameters pattern is also useful for accumulating several partial results. The Tutorial_Fibonacci module has several different examples and a discussion on creating efficient functions for computing the Fibonacci numbers. One of those efficient functions, fib2, uses three accumulating parameters. This makes sure that the state needed to efficiently perform the computation is available at each recursive step, and in a sufficiently evaluated form. It is analogous to how such a Fibonacci function would be efficiently implemented in Java, except that the use of mutable local state variables in Java is replaced by the accumulating parameters. The CAL version is much more declarative and equally as efficient as the Java version.

```
fib2 :: Int -> Integer;
fib2 !n =
    let
        /**
         * Computes the nth fibonacci number
         * (where n is defined in the enclosing scope
         *  as the argument to fib2).
         * @arg i an index (will be <= n)
         * @arg fib_i_minus_1 the (i-1)th fibonacci number
         * @arg fib_i the ith fibonacci number
         * @return the nth fibonacci number
         */
```

```
        fibHelper :: Int -> Integer -> Integer -> Integer;
        fibHelper !i !fib_i_minus_1 !fib_i =
            if i == n then
                fib_i
            else
                fibHelper
                    (i + 1) fib_i (fib_i_minus_1 + fib_i);
    in
        fibHelper 1 0 1;
```

8. What is the Prelude.eager function and when would I use it?

Expressions in CAL can be compiled in one of two ways: strictly or lazily.

Compiling an expression strictly happens when the compiler determines that the expression needs to be evaluated to Weak Head Normal form when its enclosing expression is evaluated to WHNF. If this is not the case, then the expression will be compiled lazily, which means that a suspension or thunk will be created encapsulating the state required to evaluate the expression to WHNF later if this is needed as a result of subsequent evaluation.

The Prelude.eager function causes the compiler to compile its argument expression strictly, overriding the default behavior described above. In the case where the expression would be compiled strictly anyways, it has no effect whatsoever.

There are a variety of situations in which it is handy to use the Prelude.eager function.

a. If an expression will certainly be evaluated to WHNF it is more efficient to compile the expression strictly, rather than compile the expression lazily, and then later evaluate the lazy suspension to WHNF. This is because when the suspension is eventually evaluated, it will, at best, require the same amount of work as evaluating the expression to WHNF would in a strict compilation scheme in the first place. So the cost of creating the lazy suspension itself represents an extra overhead. This cost can be thought of as basically the cost of allocating memory for an object representing the suspension, and initializing its fields.

b. If an expression may or may not be evaluated to WHNF, but the cost of actually doing the work of evaluating the expression, even if the result is not used, is less than the cost of allocating a suspension to compute the expression lazily, then one might as well evaluate the expression eagerly.

In both these cases, it is important to take into account the fact that evaluating an expression eagerly changes the order of evaluation, so special care must be taken if this issue is important, such as if the expression being evaluated has side effects or may terminate in an exception.

9. How do I create an abstract data type in CAL?

An *abstract data type* is one whose internal representation can be changed without needing to modify the source code of client modules that make use of that type. For software maintainability, it is a good idea to make a type that is subject to change or enhancement into an abstract data type. Another reason to create an abstract data type is to enforce invariants for values of the type that can only be ensured by using *constructor functions* (i.e. functions that return values of that type).

In principle it is simple to create an abstract data type in CAL. For an algebraic data type, make the type constructor public and all data constructors private. For a foreign data type, make the type constructor public and the implementation scope private. If a scope qualifier is omitted, the scope is taken to be private.

For example, the `Map` algebraic data type has the public type constructor `Map` and the data constructors `Tip` and `Bin` are each private, so it is an abstract data type.

```
/** A map from keys (of type {@code k@}) to values
   (of type {@code a@}). */
data public Map k a =
    private Tip |
    private Bin
        size      :: !Int
        key       :: !k
        value     :: a
        leftMap   :: !(Map k a)
        rightMap  :: !(Map k a);
```

There are a number of invariants of this type: the `size` field represents the number of elements in the map represented by its `Bin` value. The keys in `leftMap` are all less than `key`, which in turn is less than all the keys in `rightMap`. In particular, non-empty Map values can only be created if the key parameter type is a member of the `Ord` type class.

Values of the `Map` type can be created outside the `Cal.Collections.Map` module only by using constructor functions such as `fromList` and `insert`:

```
fromList :: Ord k => [(k,a)] -> Map k a;
public fromList !list = …

insert :: Ord k => k -> a -> Map k a -> Map k a;
public insert !key value !map = …
```

The owner of the `Cal.Collections.Map` module must ensure that all invariants of the `Map` type are satisfied, but if this is done, then it will automatically hold for clients using these functions.

Some examples of foreign abstract data types are `Color`, `StringNoCase` and `RelativeDate`:

```
data foreign unsafe import jvm private "java.awt.Color"
    public Color;

data foreign unsafe import jvm private "java.lang.String"
    public StringNoCase;

data foreign unsafe import jvm private "int"
    public RelativeDate;
```

The private implementation scope for `Color` means that a foreign function whose type involves `Color` can only be declared in the `Cal.Graphics.Color` module where the `Color` type is defined. A foreign function declaration involving the `Color` type relies on the compiler knowing that the `Color` type corresponds to `java.awt.Color` to resolve the corresponding Java entity i.e. it must know about the implementation of the `Color` type. Having a private implementation scope means that the `Color` type can be changed to correspond to a different Java class, or indeed to be an algebraic type, without the risk of breaking client code.

In all these three cases there are useful, and different, design reasons to adopt a private implementation scope:

For `RelativeDate`, the Java implementation type `int` represents a coded Gregorian date value in the date scheme used by Crystal Reports. Not all `int` values correspond to valid dates, and the algorithm to map an `int` to a year/month/day equivalent is fairly complicated, taking into account things like Gregorian calendar reform. Thus, it is desirable to hide the implementation of this type.

For `StringNoCase`, the implementation is more straightforward as a `java.lang.String`. The reason to adopt a private implementation scope is to ensure that all functions involving `StringNoCase` preserve the semantics of `StringNoCase` as representing a case-insensitive string value. Otherwise it is very easy for clients to declare a function such as:

```
foreign unsafe import jvm "method replace"
    replaceChar :: StringNoCase -> Char -> Char -> StringNoCase;
```

which does not handle case-insensitivity correctly, but is a perfectly valid declaration. This declaration results in a compilation error when it is placed outside the module in which `StringNoCase` is defined because of the private implementation scope of `StringNoCase`.

For `Color`, the issue is somewhat more subtle. `The java.awt.Color` implementation type is semantically the same as the CAL `Color` type. The problem is that `java.awt.Color` is mutable (since it can be sub-classed to create a mutable type). It is preferable for a first-class CAL type to not be mutable, so we simply make the implementation scope private to ensure that this will be the case.

A somewhat less encapsulated kind of abstract data type can be created using *friend modules* and *protected* scope. For example, if an algebraic type is public, and all its data constructors are protected, then the data constructors can be accessed in the friend modules of the module in which the type is defined. Effectively this means that the implementation of the semantics of the type stretches over the module in which the type is defined, and all of its friend modules. These must all be checked if the implementation of the type is modified.

Given the merits of abstract data types discussed above, it is perhaps surprising that most of the core types defined in the Prelude module are not abstract data types. For example: `Boolean`, `Char`, `Int`, `Double`, `String`, `List`, `Maybe`, `Either`, `Ordering`, `JObject`, `JList`, and all record and tuple types are non-abstract types.

There are different reasons for this, depending on the particular type involved.

For example, `Boolean`, `List`, `Maybe`, `Either` and `Ordering` are all rather canonical algebraic data types with a long history in functional languages, with many standard functions using them. They are thus guaranteed never to change. In addition, their values have no particular design invariants that need to be enforced via constructor functions. Exposing the data constructors gives clients some additional syntactic flexibility in using values of the type. For example, they can pattern match on the values using case expressions or let patterns.

Essentially the same explanation holds for record and tuple types. Although non-tuple record types are less canonical, they do correspond to the fundamental notion of an anonymous named-field product type. The "anonymous" here simply means that the programmer can create an entirely new record type simply by creating a value; the type does not have to be declared anywhere prior to use.

`Char`, `Int`, `Double`, `String`, `JObject` and `JList` are foreign types where in fact part of the semantics of the type is that we want clients to know that the type is a foreign type. For example, we want clients to know that `Prelude.Int` is essentially the Java primitive unboxed `int` type, and has all the semantics you would expect of the Java `int` type i.e. this is quite different from `RelativeDate` which is using `int` as its implementation type in a very tactical way that we may choose to change. One can think of a public foreign type declaration with public implementation scope as simply introducing the Java type into the CAL namespace.

One interesting point here is with CAL's naming convention for public foreign types. We prefix a type name by "J" (for "Java") for foreign types with public implementation type such that the underlying Java type is mutable. This is intended as mnemonic that the type is not a pure functional type and thus some caution needs to be taken when using it. For example, `Prelude.JObject` has public Java implementation type `java.lang.Object`.

In the case where the underlying Java type is not mutable, we do not use the prefix, since even though the type is foreign; it is basically a first class functional type and can be freely used without concern. For example, `Prelude.String` has public Java implementation type `java.lang.String`.

In the case where the implementation type is private, then the fact that the type is a foreign type, whether mutable or not, is an implementation detail and we do not hint at that detail via the name. Thus `Color.Color` has as its private Java implementation type the mutable Java type `java.awt.Color`.

When creating abstract data types it is important to not inadvertently supply public API functions that conflict with the desired public semantics of the type. For example, if the type is publicly a pure-functional (i.e. immutable) type such as Color, it is important not to expose functions that mutate the internal Java representation.

A more subtle case of inadvertently exposing the implementation of a type can occur with derived instances. For example, deriving the `Prelude.Outputable` and `Prelude.Inputable` type classes on a foreign type, whose implementation type is a mutable Java reference type, allows the client to gain access to the underlying Java value and mutate it (by calling `Prelude.output`, mutating, and then calling `Prelude.input`). The solution in this case is to not derive `Inputable` and `Outputable` instances, but rather to define a custom `Inputable` and `Outputable` instance that copies the underlying values.

10. CAL programming tips

What follows is a list of CAL programming tips in note form. Some of these tips will be expanded into full essays as above in the future. In the meantime, hopefully they will be of some use even in this short form.

- follow the CAL code formatting conventions
    - JEdit makes this particularly nice
    - setting Eclipse's Java editor to recognize .cal files as Java also works OK.
- avoid the use of un-encapsulated mutable foreign types
- don't use a record or tuple type when an algebraic type will work
    - the algebraic type offers better abstraction (the name of the type) better encapsulation, better performance characteristics and better control
- be aware of what derived instances mean- both when to use them, and when not to use them
- document your code using CALDoc
    - it is important to provide CALDoc for modules, types and instances.
- unit test your functions
    - create examples functions
    - unit test for correctness as well as performance (speed, space usage, scalability)

    - follow existing examples in the libraries for how to do this.
- provide top-level type declarations, and type declarations for local functions whenever possible
- practice good code encapsulation
    - make data constructors private
    - make the implementation type of foreign types private
    - provide constructor functions that validate arguments e.g. if a certain Int field needs to be positive.
    - comment on the invariants of the type in its definition.
- understand the operational evaluation semantics of CAL- lazy reduction order
    - understand the seq function
    - understand plinging
        - plinging of data constructor fields
        - plinging of function arguments
- understand the accumulating parameter coding pattern
    - use extra arguments to cache partial computations or intermediate results
    - when writing a directly recursive function, don't perform the computation in the returned value; use an argument instead
- be aware of the performance characteristics of the library data structures you are using
    - List, String, Map, IntMap, Set, records/tuples, …
- learn and use the functions from the libraries.
- keep things as simple as possible
    - create types to increase abstraction and encapsulation
    - avoid writing overly long functions
        - they are harder to understand and unit test
    - avoid the overuse of deep levels of nesting (i.e. many levels of local function definitions)
    - avoid writing overly long modules
        - break up into several modules and take advantage of the friend module pattern and separate compilation to organize your code into comprehensible chunks.
- consider defining instances for the standard type classes if they make sense
- look for examples in the existing CAL code for how a problem was solved.
- to provide extensibility to a module's functionality,
    - use higher order functions
    - use abstract data types in the api (can later add new constructors for the data types)
    - use type classes
    - favour the use of higher order functions over type classes
- consult the existing resources on CAL
    - CAL User's Guide
    - the source code for the existing CAL modules, especially non-test modules in CAL Platform and CAL Libraries.
    - the Tutorial* .cal modules
    - generated CAL doc (this is easily generated from the GemCutter)

- be aware of the tools available for CAL development
    - GemCutter
    - ICE
    - -etc
- learn Haskell and study well-known and respected Haskell libraries
- understand how to mutate
    - when writing functions that must manipulate state within their implementation, try to encapsulate them so that the publicly exposed version is a pure stateless function.

other advice

- know how to generate lecc Java source code. Study the source of functions you create. Are they reasonable?
- learn the various ICE diagnostic tools such as call counts, logging, etc
- understand the Prelude.eager function and when to use it.
- don't get discouraged!