LES LIAISONS DANGEREUSES: IS IT AN INTERPRETER?

The Illumination of Recursion and Structure with an Arithmetic Data Type

Fundamentals of Programming

Final Project

Brian Scott Bohme

Table of Contents

Introduction

Les Liaisons Dangereuses is a stunning example of surrealist art by René Margritte. The piece shows a woman holding a painting. Inside the painting is the same woman holding the same frame, and so on. The idea is simple but the effect feels profound, pulling the viewer into a vicious cycle with seemingly no end. At the same time, the pattern itself completes the whole, which is a beautiful allegory for recursion in programming. As Computer Science culture would put it, "To understand recursion, you must first understand recursion." That was the heart of this project for me. Using a data type for arithmetic expressions and a byte-code syntax (more on that soon), I investigated four different optimizations for an arithmetic expression compiler, and made some *dangerous liaisons* with structural recursion to then implement a source-to-source transformer for each optimized compiler. Surprisingly though, this is not the Magritte we are referencing however, which I soon realized while working on the project. What is actually being referenced is "The Treachery of Images". "This is not a pipe" becomes "this is not an interpreter," despite our perception of it. Why and how will be uncovered soon enough. For now, I'll call it a happy coincidence that I saw parallels between two of his works and the project at hand. Along the way, I got to visualize how these "optimizations" *actually* improve performance through a series of rigorous tests. But to understand how all this happen, we must start from the very beginning with the data types.

Understanding the Arithmetic Expression and Byte Code Data Types

Throughout most of the project, there are two data types that will be the most important to understand: the source syntax (arithmetic expression) and the target syntax (byte code instruction). The source syntax is a structural representation of arithmetic. We have natural

numbers, represented as *Literal* of an integer, alongside *Plus* and *Times* of an arithmetic expression pair for addition and multiplication respectively. Therefore, the arithmetic expression representation of *"3 * (6 + 2)"* would be:

```
Times (Literal 3, Plus (Literal 6, Literal 2).
```

This is easily verified by the handy "show" function built into the source syntax module, which translates arithmetic expressions into a string like above. The "show" function is a soft version of an interpreter. The source interpreter module provides a more thorough interpretation of arithmetic expressions by evaluating the results and producing a single string result (e.g. "42") or an arithmetic expression (e.g. *Literal 42).* To test the interpreter, and all the proceeding codes throughout the assignment, there is a source sample module which is populated with some different arithmetic examples (whose pertinence is seen more clearly with some tests than others). These samples serve as a basis, but it will eventually be more efficient to generate random samples to test compilers, the de-compiler and transformers. We will see this in detail later in the project.

Now that we've worked through the source syntax, we move on to the target syntax, the arguably more complicated of the two data types. The target syntax is built as:

```
module Target_syntax : TARGET_SYNTAX =
  struct
    type byte_code_instruction = Push of int | Add | Mul
    type byte_code_program = Pro of byte_code_instruction list
  end;;
```

First is a byte code instruction. There three basic elements from arithmetic expressions are still here—integers, addition or multiplication—but we can see that both add and multiply are no longer represented as a pair. And why is it *Push* of *int?* First, we must understand that these

instructions are implemented in a byte code program, which is a list of these instructions. This is the key to understand what these byte code instructions are doing, and how they translate to an arithmetic expression. Let us look at a very simple example of a byte code program:

```
[Push 1; Push 2; Add]
```

This was obtained by compiling the arithmetic expression *Plus (Literal 1, Literal 2)* (more about compiling in the next section). It is clear now that *Add* is adding the two previous elements, which are "pushed" in order into the add environment. When constructing more complicated byte code programs, we must remember to 1) traverse the byte code program in order, and 2) "push" elements in order. This will be vital when implementing a de-compiler. First, let's focus on the compiler, and how to translate from source syntax to the target syntax.

The Magritte Interpreters

Throughout the project, we will be testing each compiler's commutation with Magritte interpreters. But what are they doing exactly? First, let's look at the standard interpreter to compare.

```
module Source_interpreter_standard : SOURCE_INTERPRETER =
  Source_interpreter_maker (Natural_number_semantics);;


module Source_interpreter_maker (Natural_number : NATURAL_NUMBER) :
SOURCE_INTERPRETER =
  struct
    let rec evaluate e =
      (* evaluate : Source_syntax.arithmetic_expression ->
Natural_number.nat *)
```

```
      match e with
        Source_syntax.Literal n ->
         if n < 0
         then raise (Negative_integer n)
         else Natural_number.quote n
      | Source_syntax.Plus (e1, e2) ->
         Natural_number.plus (evaluate e1, evaluate e2)
      | Source_syntax.Times (e1, e2) ->
         Natural_number.times (evaluate e1, evaluate e2)

    let interpret_to_string e
      = Natural_number.string_of_nat (evaluate e)

    let interpret_to_source e
      = Natural_number.syntax_of_nat (evaluate e)
  end;;

module Natural_number_semantics : NATURAL_NUMBER =
  struct
    type nat = int

    let quote n =
      n

    let plus (n1, n2) =
```

```
      n1 + n2

    let times (n1, n2) =
      n1 * n2

    let string_of_nat n
      = string_of_int n

    let syntax_of_nat n =
      Source_syntax.Literal n
  end;;
```

The first thing I notice is that the natural number semantics module is actually evaluating the expressions mathematically. Therefore I can instantly deduce that the output of the standard interpreter should be a single value (Literal if in source syntax, String of integer if in string syntax). And it is easy to see that the evaluate function is doing exactly that; taking an arithmetic expression, and producing a natural number using the natural number semantics. When we move onto the Margritte interpreter, we notice something else happening:

```
module Source_interpreter_Magritte : SOURCE_INTERPRETER =
  Source_interpreter_maker (Natural_number_syntax);;

module Natural_number_syntax : NATURAL_NUMBER =
  struct
    open Source_syntax
```

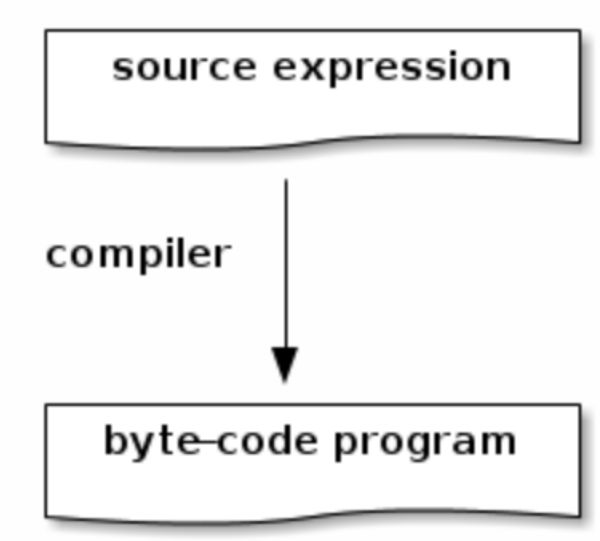
```
    type nat = arithmetic_expression

    let quote n =
      Literal n

    let plus (e1, e2) =
      Plus (e1, e2)

    let times (e1, e2) =
      Times (e1, e2)

    let string_of_nat e =
      show e

    let syntax_of_nat e =
      e
  end;;
```

The natural number syntax is not actually evaluating the natural numbers. Instead, we get a syntactical representation of the expression, or an unparser, rather than a fully evaluating interpreter. The “Just In Time” modules confirm this with “the compile and convert to string” functions; while the standard interprets the expression to single integer result as a string, the Magritte preserves the entire expression, instead producing a string representation of the syntax. *Ce n'est pas un interprète*! We are interpreting the arithmetic expression, yet we are not! The

advantage of this may not be clear yet, but when we start implementing source-to-source transformation, Magritte interpreters play an important role in understanding the optimizations at hand.

The Primitive Compiler, and the First Optimization

The compiler is the first key part to this project. As mentioned in the previous section, our compiler should take an arithmetic expression as an input, and output the corresponding byte code program. Diagrammatically:



We were given the original compiler, and told to write an accumulator-based version of it. I wanted to understand what was happening in the primitive compiler first however:

```
let compile e_init =
    let rec flatten e =
        match e with
          Source_syntax.Literal n ->
           if n < 0
```

```
            then raise (Negative_integer n)
            else [Target_syntax.Push n]
          | Source_syntax.Plus (e1, e2) ->
            flatten e1 @ flatten e2 @ [Target_syntax.Add]
          | Source_syntax.Times (e1, e2) ->
            flatten e1 @ flatten e2 @ [Target_syntax.Mul]
        in Target_syntax.Pro (flatten e_init)
```

A *Literal n* becomes *Push n*, as expected. We can really see the construction of the byte code instruction program list with the *Plus* and *Times* match cases. Notice how the "push" of list elements into add/multiply environments we noted in the previous section is clearly elucidated here. The first element becomes prepended to the second, which is appended by the *Add* or *Mul* byte code instruction. Since the construction of this is so fluid and intuitive with the recursion, building an accumulating version was simple. First, we must include an accumulator, which will be initialized as an empty list. Encountering a *Literal* should just be prepending *Push* to the accumulator. Thus, *Literal n* becomes *[Push n]*, which is consistent with the byte code syntax. For the other two cases, we want to make sure the first expression in the pair is always prepended to the second, followed by the *Add/Mul* instruction. Preserving this order is the most important part!! The instruction should be prepended to the accumulator, or else (since Literals are prepended as well) eventually order will not be correct. We then want to flatten the second element of the pair to the accumulator, which becomes the accumulator for flattening the first element of the pair. Therefore, we have e1::e2::[instruction]::acc, all in proper byte code syntax. Syntactically, this looks like:

```
let compile e_init =
    let rec flatten e a =
```

```
        match e with
        | Source_syntax.Literal n ->
            if n < 0
            then raise (Negative_integer n)
            else (Target_syntax.Push n)::a
        | Source_syntax.Plus (e1, e2) ->
            flatten e1 (flatten e2 ((Target_syntax.Add)::a))
        | Source_syntax.Times (e1, e2) ->
            flatten e1 (flatten e2 ((Target_syntax.Mul)::a))
    in Target_syntax.Pro (flatten e_init []).
```

Talk about recursion for the sake of recursion! When testing if the accumulator compiler is actually outputting the same result as the regular compiler, I just created a function that tests if the output of each compiler is the same for a random expression, and repeated it *n* times (10000 in this case). I also parametrized the accumulating compiler so I could test my fold-right version as well! This way, no expensive mapping or intermediate lists are necessary, making it more efficient and flexible with testing.

But how much more efficient is an accumulator-based version? No better way to test this than a race! Because both compilers can run small tests very quickly, it is hard to discern a noticeable difference. But for an *extremely large* test, the discrepancies in speed are undeniable! First, I implemented a random expression generator using a random integer match similar to the previous mini-project with logic expressions. I put a safety of 200 on the random expression generator, meaning each sub-expression can be a maximum of 201 expressions long. Then, I generated a test list by appended 10000 of these expressions together. Then, we map each function over the same list (therefore not needing to generate two different lists for each). By

mapping each compiler over the lists, we can already see that the non-accumulating compiler is a lot slower. But it is much more fun to see exactly how much slower. I built a fancy timer to test how long each compiler takes to execute the program using the Unix module, taking the time before and after the compiler compiles every expression in the list:

```
let test_of_speed compiler =
    let start = Unix.gettimeofday ()
    in let _result = List.map (fun l -> compiler l)
                              (test_list_of_arithmetic_expressions)
    in let stop = Unix.gettimeofday ()
    in Printf.printf "Done!\nExecution time: %fs\n%!"
      (stop -. start);;
```

The results are in, and the winner is… the accumulating compiler by a huge margin! In fact, the accumulating compiler is able to cruise through the list in just .16 seconds, while the primitive compiler took between 5-8 seconds per test and sometimes even blew the stack. Why? Because it is tail recursive! It is compiling on the go, rather than waiting until all the recursions are done, then compiling everything, then appending the lists together. Every time we implement an accumulator, the advantage is that we are *accumulating* (i.e. constantly computing) the result, instead of computing it at the end. With our gold medalist in compiling crowned, we're able to move onto the next bit: de-compiling.

Let's Reverse! Or: The De-compiler

The goal of the de-compiler is to input the target syntax and output the source syntax. It is important to note that the source expression, when compiled then decompiled, will not always be the same when outputted. Depending on how the compiler compiles the source expression, the

de-compiler may output an optimized arithmetic expression. We'll see how and why these expressions are optimized in the subsequent sections. Of course, when evaluated the input should not differ from the output when evaluated (for example, the result of e120 should not be different from the result of compiled then de-compiled e120 when evaluated).

In the previous section, I talked a lot about the importance of keeping byte instructions in a certain order, and how byte code Literals are "pushed" into the arithmetic instructions (plus and times). What this immediately tells me is that we need some kind of ordered data structure to keep track of the byte code instructions. We saw that when traversing the byte code instruction list, it should always be the case that the two most recent byte code instructions are being pushed into the plus or times environment:

[Push x; Push y; Push z; Add…]

becomes

[Push x; Add (Literal y, Literal z)…].

But obviously we can't have the source syntax within a target syntax list, so an accumulator is necessary. This is when I knew a stack would be the perfect data type; because it "last-in first-out", it correctly maintains the necessary order and is the least expensive to access (because you are only accessing the heads of the list, not the last elements). The byte code instruction list is therefore a queue, where the first element is the first pushed out, into the stack. Therefore, in theory the above example should look like:

Queue: [Push x; Push y; Push z; Add; Times…]	Stack: []
Queue: [Push y; Push z; Add; Times…]	Stack: [Literal x]
Queue: [Push z; Add; Times…]	Stack: [Literal y; Literal x]
Queue: [Add; Times…]	Stack: [Literal z; Literal y; Literal x]
Queue: [Times…]	Stack: [Add (Literal y, Literal z); Literal x]

Queue: […] Stack: [Times (Literal x, Add (Literal y, Literal z)]

In the end, there is only one element in the stack, which is the corresponding (and correct) source expression from the original byte code instruction list. What this tells us is: 1) any "push n" instruction should be pushed to the stack as a "Literal n" and 2) any time an "add" or "times" is recursed upon in the queue, the first 2 elements of the stack should be the corresponding pair. This arithmetic expression should be pushed to the top of the stack. The corresponding code follows this exact structure. One of the interesting things about the code is the first match case:

match e with

```
| [] ->
    (match e_acc with
     | e_sub_decompiled::stack ->
              e_sub_decompiled
     | _ ->
             raise Stop_everything).
```

Why should a stack with no elements or multiple elements return an error or just the top of the stack respectively? The empty stack is more obvious: there are no byte code instructions inputted! Everything is pushed to the stack, so if there is nothing there in once the queue is empty, then there were no instructions to begin with. The second case is more nuanced. Because the stack is actively transforming byte code instructions (most importantly addition and multiplication) to the source syntax, theoretically there should be only 1 element in the stack in the end. If there is more than 1 element in the end, that means there was a bad byte code instruction list. An example of this would be:

[Push 1; Push 2: Add; Push 3].

It is clear that the "Push 3" is not arithmetically connected to the addition of 1 and 2. If we were

to interpreter this with the target syntax Margritte interpreter without error, it'd look something like:

"(1 + 2) 3" ,

with the 3 completely disconnected from the (1+2). It is clear from the de-compiler that this result in a 2-element stack, which should raise an error.

To test the de-compiler, we want to make sure that a compiled then de-compiled expression is the same as the source expression (since we aren't using an optimizing compiler yet, this should always be true). We can use our random arithmetic expression generator to test this with a bunch of different expressions, using the test code:

```
let rec test_compile_then_decompiled n =
    let expression = random_arithmetic_expression_generator 50 in
    let test =
           (Decompiler.decompile (Compiler.compile expression) =
           expression)
    in
    if n = 0 then test
    else
    test && test_compile_then_decompiled (n-1);;
```

where *n* is the number of times we want to run the test with different arithmetic expressions. With the de-compiler in order, we can move on to the optimizing compilers.

The Surprising Compiler

To see what optimization this compiler is making, we just need to compile then de-compile a couple of expressions. Using the test expressions from Source_sample:

```
e0’ = Source_syntax.Times
(Source_syntax.Times
   (Source_syntax.Times (Source_syntax.Literal 0,
Source_syntax.Literal 1),
   Source_syntax.Times (Source_syntax.Literal 2, Source_syntax.Literal
3)),
 Source_syntax.Times (Source_syntax.Literal 4, Source_syntax.Literal
5)).
Decompiler.decompile (Compiler_surprising.compile e0’) = Literal 0
_____
e6’’ = Source_syntax.Plus
(Source_syntax.Plus (Source_syntax.Literal 0, Source_syntax.Literal
1),
 Source_syntax.Plus (Source_syntax.Literal 2, Source_syntax.Literal
3))

Decompiler.decompile (Compiler_surprising.compile e3) =
Source_syntax.Plus (Source_syntax.Literal 1,
Source_syntax.Plus (Source_syntax.Literal 2, Source_syntax.Literal 3))
_______
e2 = Source_syntax.Times (Source_syntax.Literal 1,
Source_syntax.Literal 2)

Decompiler.decompile (Compiler_surprising.compile e2) =
Source_syntax.Literal 2;;
```

With these examples, we can notice that the compiler simplifies expressions that add 0 or multiply by 1 or 0. Mathematically, any expression multiplied by 1 or added to 0 is itself, while any expression multiplied by 0 is 0 (Literal 0 in source syntax). We can see this in work when passed through the surprising compiler with simple examples: for example, if our expression is Plus (Literal 0, Literal 1), we first recurse on the "Literal 0", which leads us to the function:

```
visit
(Literal 1)
(fun () ->  [Target_syntax.Push 0])
(fun () ->  [Target_syntax.Push 1])
(fun bcis -> bcis).
```

The match case on Literal 1 invokes:

```
(fun () ->  [Target_syntax.Push 1]) (),
```

which outputs,

```
[Target_syntax.Push 1],
```

which is the result we want.

More complicated expressions build off this same structure and logic, but the important feature bolstering the compiler is the different functions passed through as k0 (function invoked if Literal 0), k1 (function invoked if Literal 1), and k (utilizes hammer and weld to build and modify the accumulating list of byte code instructions when k is not 0 or a multiplication by 1).

Therefore, our source-to-source transformer should output arithmetic expressions with no multiplications by 1 or 0, and no additions by 0. The most obvious thing to do is a match case, with 0 (for plus) and 0 and 1 (for times). Therefore, the appropriate optimization can be made for each. However, the most important thing to remember is that the most efficient route would be to

start from the inner-most expressions (Literals), and work our way out. Therefore, for example, if a “Times (Literal 0, _)” is encountered, we know that we can just produce “Literal 0”. By matching upon the recursed expressions rather than implementing recursion on each expression in order, we can also make it structurally recursive (because we wouldn’t be able to access a parent expression to a subexpression if we’ve already iterated over it without inputting the whole expression again). The match cases are very straight forward, as we are just doing exactly as described above:

```
let transform e_init =
      let rec normalize e =
        match e with
        | Literal n ->
            e
        | Plus (e1, e2) ->
            (match (normalize e1), (normalize e2) with
              | Literal 0, e2' ->
                      e2'
              | e1', Literal 0 ->
                      e1'
              | e1', e2' ->
                      Plus (e1', e2'))
        | Times (e1, e2) ->
            (match (normalize e1), (normalize e2) with
              | Literal 0, _ ->
                      Literal 0
```

```
              | _ , Literal 0 ->
                      Literal 0
              | e1', Literal 1 ->
                      e1'
              | Literal 1, e2' ->
                      e2'
              | e1', e2' ->
                      Times (e1', e2'))
     in normalize e_init.
```

Since this is structurally recursive (we are never implementing recursion over a larger subexpression than the preceding expression), it is easy to implement this as a fold right expression:

```
let fold_right_arithmetic_expressions lit add mul e_init  =
    let rec visit e =
      match e with
      | Source_syntax.Literal n -> lit e
      | Source_syntax.Plus (e1, e2) -> add (visit e1) (visit e2)
      | Source_syntax.Times (e1, e2) -> mul (visit e1) (visit e2)
    in visit e_init;;


let transformer_surprising_using_fold_right e_init =
    fold_right_arithmetic_expressions
    (fun n -> n)
```

```
    (fun e1 e2 -> (match e1, e2 with
                    | Source_syntax.Literal 0, e2' -> e2'
                    | e1', Source_syntax.Literal 0 -> e1'
                    | e1', e2' -> Source_syntax.Plus (e1', e2'))
    )
    (fun e1 e2 -> (match e1, e2 with
                    | Source_syntax.Literal 0, _ ->
Source_syntax.Literal 0
                    | _ , Source_syntax.Literal 0 ->
Source_syntax.Literal 0
                    | e1', Source_syntax.Literal 1 -> e1'
                    | Source_syntax.Literal 1, e2' -> e2'
                    | e1', e2' -> Source_syntax.Times (e1', e2'))
    )
      e_init;;
```

We can see that this passes the test of commutativity (the source to source transformation is the same as compiling then decompiling the source expression) and idempotence (running the source to source transformation yields the fully optimized expression in the first pass through, such that subsequent inputs into the transformer always yield the same result). We can further test the source to source transformer by implementing a syntax checker. The syntax checker should essentially make sure that no additions by 0 or multiplications by 0 or 1 occur in the transformed expression. If we simply iterate through, and output false if either of these conditions is violated, then we've got a simple yet effective syntax checker!

The true question is how optimized the source-to-source transformer is compared to compiling then decompiling the expression. Logic tells me that the source-to-source transformer should be much faster; we are never compiling to the target syntax, and aren't passing through the expression as many times as compiling then decompiling. To really test it though, of course I'm going to implement a race again. With both contestants clocked in, the winner is (as expected) the transformer, which proves to be 30 times faster when tested on a list of 10000 arithmetic expressions.

The Bizarre Compiler

I found the bizarre compiler much more difficult to express as a structurally recursive source-to-source transformer than the surprising compiler. Unfortunately, I haven't been able to implement a fully structurally recursive transformer yet. However, I am able to explain the optimization behind the compiler and implement a functional transformer, and my thought process behind a structurally recursive version of the transformer.

What I immediately noticed that expression compiled by the bizarre compiler then de-compiled, if parsed into a string with the Margritte interpreter, associated all multiplication and/or addition to the tail of an expression. Mathematically:

"(1 + 2) + 3" would become "1 + (2 + 3)",

or

"(1 + 2) + (3 + 4)" would become "1 + (2 + (3 + 4))",

and so on and so forth. Even though there are no parenthesized values in arithmetic source syntax per se, the order of operations bounded by parenthesis is still extremely (and vitally) relevant in the source syntax. When evaluated, it is imperative to evaluate the subexpressions in

the same order and manner as if they were parenthesized (inner-most to outer-most). We can see this in action with the compiler. The compiler has 3 different functions for Literal, Add and Times expressions. When consecutive Add or Times constructors are inputted, a specific order in which expressions become appended to the accumulator is implemented so when decompiled they are in the optimized syntax:

```
visit_plus e k a =
          match e with
            Source_syntax.Literal n ->
             if n < 0
             then raise (Negative_integer n)
             else k [Target_syntax.Push n] a
          | Source_syntax.Plus (e1, e2) ->
             visit_plus e1
                        (fun bcis1 a1 ->
                          append bcis1 (visit_plus e2 k a1))
                        (Target_syntax.Add :: a)
```

This is part of the plus function from the compiler. We can see how the second expression is going to be parsed first, prepended by the first. However, the most important part is the all the Add constructors are being appended to the Literals with consecutive Pluses. This is how, for example, the output:

[Push 1; Push 2; Push 3; Add; Add]

is obtained from:

Plus (Plus (Literal 1, Literal 2), Literal 3)).

This is why the auxiliary append function comes in handy, because we want to keep the add expressions at the end of the byte code instruction list (for each subexpression).

I first implemented a straightforward version of the source to source transform in an attempt to try and understand the structural recursion behind the transformer. What I noticed is that:

Plus (Plus (e1, e2), e3) becomes Plus (e1, Plus (e2, e3)),

and likewise for multiplication. I directly implemented this into a match case. However, the non-structural recursion part is that you have to recurse over the entire expression, because the extra "Plus" constructer may have introduced a new case of un-optimized sub-expressions. Therefore, while straightforward and easy to read, this source to source transformer isn't structurally recursive.

I spent hours trying to make this transformer structurally recursive, to no avail. My main point of reference was the midterm project, where I tackled a very similar form of structural recursion. In a similar fashion, there was a modifying constructor (in that case Negation) that had to be carried through the recursion without actually being recursed upon. In that case, I made the normalizer a function of polarity, which was a true/false switch that determined if the Variable should be negated or not in the end. However, the trouble I have with applying that logic to this transformer is the modifying constructor affects a) two subexpressions and b) not implemented upon the base Literal, but rather the subexpressions of a Plus/Times constructor. I had a hard time reconciling this, because a simple "switch" like the negational normalization wouldn't keep track of the number of expressions to optimize (because it's not a binary like negation). Likewise, I couldn't figure out when to actually implement the optimization. With the negational normalization, the final optimization happened with the Variable, which was straightforward.

Here, once you reach a Literal it is too late to access the other subexpressions in order to make the optimization.

I tried a couple of different strategies to try and crack the structural recursion. First, I made an accumulator to collect values that would be stored until the consecutive add/times expressions were iterated through, in which they'd be distributed as described above. However, this failed to fully optimize the expression. It'd usually take 2-3 passes through the function to become idempotent, which means it wasn't a correct implementation of a source to source transformer. I tried doing the same accumulating strategy with multiple accumulators for each subexpression, but this resulted in many duplicates of the same subexpression. I'm hoping soon I'll "see it" and it'll all click, but unfortunately this task is proving to be much, much less intuitive than the surprising source to source transformer and even the negational normalization task.

Despite these setbacks, I was still able to test my non-structurally recursive transformer for commutation and idempotence, in which it passed both tests. I also made a syntax checker, which (very similarly to the surprising complier) would output false if consecutive "Plus" or "Times" constructers were found in the syntax (e.g. Plus (Plus (e1, e2), e3) or Times (Times (e1, e2), e3)), and verified this using the same methods as the surprising compiler. Again, this transformer proved to be 30 times faster than the compiled-decompiled method with the same list of 10000 arithmetic expressions. Because it isn't taking the intermediate step of compiling into the target language and back again, it is evident why this is the case.

Further Optimizations Ahead

The final compiler in this assignment was the quaint compiler, which was easy to understand what it was doing but extremely difficult to actually implement as an efficient and

structurally recursive transformer. Distributes multiplication across subexpressions, such that the product of summands (e.g. Times (Plus (Literal 1, Literal 2), Literal 3) becomes the sum of factors (e.g. the prior example becomes: Plus (Times (Literal 1, Literal 3), Times (Literal 2, Literal 3))). If these expressions were unparsed using the Margritte interpreter, this would show as a distribution of multiplication inside parenthesized values. However, it is obvious that the output of the source to source transformer should, for the most part, be larger than the source expression. How to make this structurally recursive was something I could not reconcile, even after hours of drawing the recursion out by hand. The same issue plagued me as before with the bizarre transformer: when to actually implement the optimization (because you can't do it once you reach the Literal values), and how to keep track of and implement the number of consecutive optimizations occurring. As is, it took multiple passes of the expression to finally become idempotent unfortunately. These last two transformers really made me put a lot of thought in, and unfortunately it came without a lot of fruition.

It would be interesting to see if you could implement a structurally recursive transformer with a coinciding compiler that could implement all of these optimizations, without just passing the expression through all three transformers/compilers. I imagine it would be quite tedious (especially the transformer), because you are having to juggle three different optimizations with one pass through, simultaneously modifying the subexpressions alongside the whole arithmetic expression. It would be an interesting challenge, but first I'd have to firmly implement all three of the transformers to even get started on it!

Parting Thoughts

I'm very disappointed that I couldn't fully implement a structurally recursive transformer for the bizarre compiler. I spent hours poring over it, yet the answer wouldn't click with me. I'm

happy that I was able to figure out the syntactical backing for it however, and implement a strong de-compiler and “surprising” transformer. Throughout the midterm project and this final project I learned a lot about structural recursion and implementing data types within a modular setting. Both projects really challenged me to really think about how structure and syntax work programmatically, and I feel more confident about my functional programming abilities because of it. Obviously I still have work to do, but at least I understand the concepts and motivations, which I see as the first step. I know I’m missing just a small part of the bizarre transformer that will come to me eventually or with a little help, and it will be a triumphant day once I can finally *completely* complete it.