Exploring Languages with Interpreters and Functional Programming Chapter 42

H. Conrad Cunningham

29 November 2018

Contents

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-5358

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of November 2018 is a recent version of Firefox from Mozilla.

42 Calculator: Abstract Syntax & Evaluation

42.1 Chapter Introduction

TODO: Check introduction, what next, acknowledgements, references, and terms once ELI Calculator chapters are complete.

The previous chapter introduced formal concepts related to concrete syntax and gave two different concrete syntaxes for the ELI Calculator language.

This chapter introduces the concepts related to abstract syntax and language semantics. It encodes the essential structure of any ELI Calculator expression as a Haskell algebraic data type and defines the semantics operationally using a Haskell evaluation function. The abstract syntax also enables the expression to be transformed in various ways, such as converting it to a simpler expression while maintaining an equivalent value.

42.2 Abstract Syntax

The *abstract syntax* of an expression seeks to represent only the essential aspects of the expression's structure, ignoring nonessential, representation-dependent details of the concrete syntax [Sestoft 2012] [Wikipedia 2018b].

For example, parentheses represent structural details in the concrete syntaxes given in the previous chapter. This structural information can be represented directly in the abstract syntax; there is no need for parentheses to appear in the abstract syntax.

We can represent arithmetic expressions conveniently using a tree data structure, where the nodes represent operations (e.g. addition) and leaves represent values (e.g. constants or variables). This representation is called a *abstract syntax tree* (AST) for the expression.

42.2.1 Abstract syntax tree data type

In Haskell, we can represent an abstract syntax trees using algebraic data types. Such types often enable us to express programs concisely by using pattern matching.

For the ELI Calculator language, we define the `Expr` algebraic data type—in the Abstract Syntax module (`AbSynCalc`)—to describe the abstract syntax tree.

```haskell
import Values ( ValType, Name )

data Expr = Add Expr Expr
          | Sub Expr Expr
```

```
            | Mul Expr Expr
            | Div Expr Expr
            | Var Name
            | Val ValType
              -- deriving Show?

instance Show Expr where
    show (Val v)   = show v
    show (Var n)   = n
    show (Add l r) = showParExpr "+" [l,r]
    show (Sub l r) = showParExpr "-" [l,r]
    show (Mul l r) = showParExpr "+" [l,r]
    show (Div l r) = showParExpr "/" [l,r]

showParExpr :: String -> [Expr] -> String
showParExpr op es =
    "(" ++ op ++ " " ++ showExprList es ++ ")"

showExprList :: [Expr] -> String
showExprList es = Data.List.intercalate " " (map show es)
```

Above in type `Expr`, the constructors `Add`, `Sub`, `Mul`, and `Div` represent the addition, subtraction, multiplication, and division, respectively, of the two operand subexpressions, `Var` represents a variable with a name, and `Val` represents a constant value.

Note that this abstract syntax is similar to the (Lisp-like) parenthesized prefix syntax described in the previous chapter.

We make type `Expr` an instance of class `Show`. We do not derive or define an instance of the `Eq` class because direct structural equality of trees may not be how we want to define equality comparisons.

We can thus express the example expressions from the Concrete Syntax chapter as follows:

```
Val 3                      -- 3
Val (-3)                   -- -3
Var "x"                    -- x
Add (Val 1) (Val 1)        -- 1+1
Add (Var "x") (Val 3)      -- x + 3
                           -- (x + y) * (2 - z)
Mul (Add (Var "x") (Var "y")) (Sub (Val 2) (Var "z"))
```

Figures 42-1 and 42-2 show abstract syntax trees for two example expressions above.

In a subsequent chapter on parsing, we develop parsers for both the prefix and infix syntaxes. Both parsers construct abstract syntax trees using the algebraic

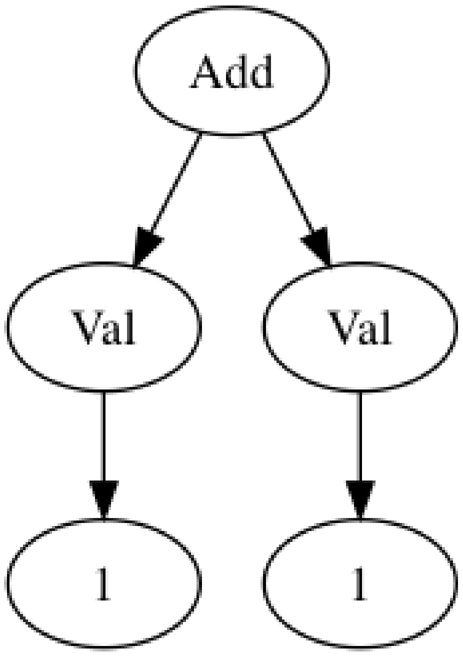

Figure 42-1: Abstract syntax tree for `1 + 1` and `(+ 1 1)`

data type `Expr`.

42.2.2 Values and variable names

The ELI Calculator language restricts values to `ValType`. The `Values` module indirectly defines this type synonym to be `Int`.

The abstract syntax allows a name to be represented by any string (i.e. type alias `Name`, which is defined to be `String` in the `Values` module). We likely want to restrict names to follow the usual "identifier" syntax. The parser for the concrete syntax should enforce this restriction. Or we could define Haskell functions to parse and construct identifiers, such as the functions below.

```haskell
import Data.Char ( isAlpha, isAlphaNum )

getId :: String -> (Name,String)
getId []          = ([],[])
getId xs@(x:_)
    | isFirstId x = span isRestId xs
    | otherwise   = ([],xs)
where
```

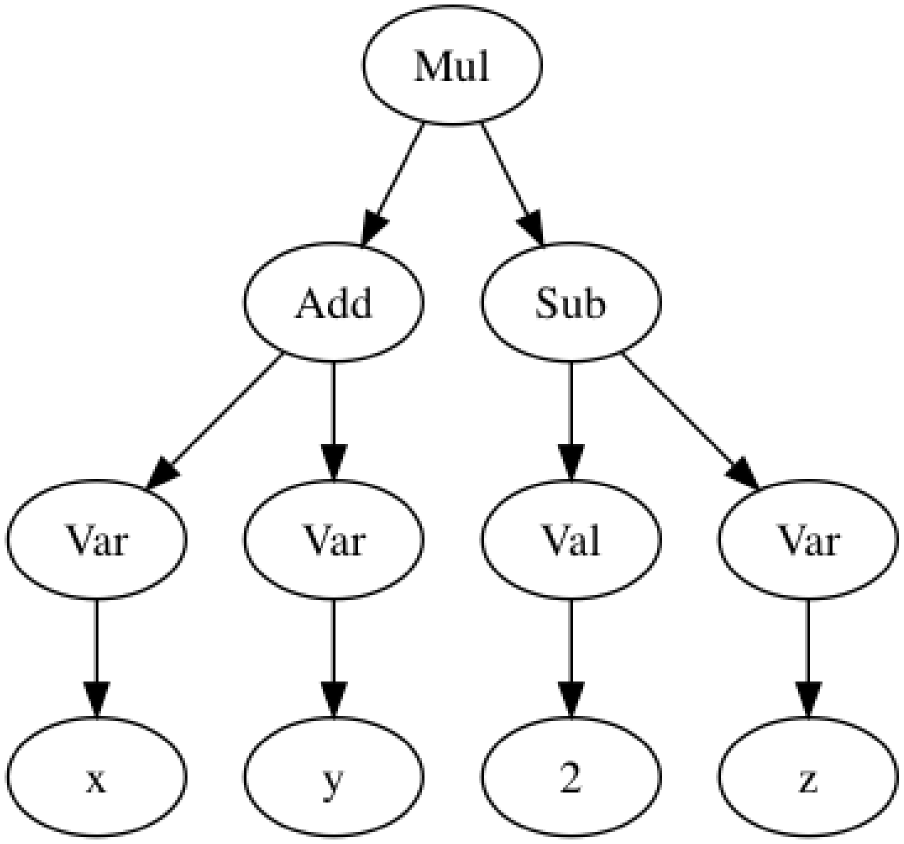

Figure 42-2: Abstract syntax tree for `(x + y) * (2 - z)` and `(* (+ x y) (- 2 z))`

```
        isFirstId c = isAlpha c    || c == '_'
        isRestId  c = isAlphaNum c || c == '_'

    identifier :: String -> Maybe Name
    identifier xs =
        case getId xs of
            (xs@(_:_),[]) -> Just xs
            otherwise     -> Nothing
```

The `getId` function takes a string and parses an identifier at the beginning of the string. A valid identifier must begin with an alphabetic or underscore character and continue with zero or more alphabetic, numeric, or underscore characters.

The `getId` function uses the higher order function `span` to collect the characters that form the identifier. This function takes a predicate and returns a pair, of which the first component is the prefix string satisfying the predicate and the second is the remaining string.

In the following chapter, we examine how to parse an expression's concrete syntax to build an abstract syntax tree.

42.3 Associative Data Structures

In language processing, we often need to associate some key (e.g. a variable name) with its value. There are several names for this type of data structure—*associative array* [Wikipedia 2018b], *dictionary*, *map*, *symbol table*, etc.

As we saw in Chapter 21, an *association list* is a simple list-based implementation of this concept. It is a list of pairs in which the first component is the *key* (e.g. a string) and the second component is the *value* associated with the key.

The Prelude function `lookup`, shown below (and in Chapter 21), searches an association list for a key and returns a `Maybe` value. If it finds the key, it wraps the associated value in a `Just`; if it does not find the key, it returns a `Nothing`.

```
    lookup :: (Eq a) => a -> [(a,b)] -> Maybe b
    lookup _    []    =  Nothing
    lookup key ((x,y):xys)
        | key == x   =  Just y
        | otherwise =  lookup key xys
```

For better performance with larger dictionaries, we can replace an association list by a more efficient data structure such as a `Data.Map.Map`. This structure implements the dictionary structure as a size-balanced tree. It provides a `lookup` function with essentially the same interface.

Of course, imperative languages might use a mutable *hash table* to implement a dictionary.

42.4 Semantics

Consider the evaluation of the ELI Calculator language abstract syntax trees as defined above.

42.4.1 Environments

To evaluate an expression, we must determine the current value of each variable occurring in the expression. That is, we must evaluate the expression in some *environment* that associates the variable names with their values.

For example, consider the expression `x + 3`. It might be evaluated in an environment that associates the value `5` with the variable `x`, written `{ x -> 5 }`. The evaluation of this expression yields the value `8`.

The environment `{ x -> 5 }` can be expressed in a number of ways in Haskell. Here we choose to represent it as a simple association list as follows:

```
[("x",5)]
```

This list associates a variable name in the first component with its integer value in the second component.

Looking up a key in an association list is an `O(n)` operation where `n` denotes the number of key-value pairs.

As noted above, a good alternative to the association list is a `Map` from the `Data.Map` library. It implements the dictionary as an immutable, size-balanced tree, thus its `lookup` function is an `O(log n)` operation.

In the ELI Calculator language implementation, we encapsulate the representation of the environment in the `Environments` module. This module exports the following type synonym and functions:

```
type AnEnv a =[(Name,a)]

newEnv     :: AnEnv a
toList     :: AnEnv a -> [(Name,a)]
getBinding :: Name -> AnEnv a -> Maybe a
hasBinding :: Name -> AnEnv a -> Bool
newBinding :: Name -> a -> AnEnv a -> AnEnv a
setBinding :: Name -> a -> AnEnv a -> AnEnv a
bindList   :: [(Name,a)] -> AnEnv a -> AnEnv a
```

For the purposes of our evaluation program, we can then define a specific environment with the type synonym `Env` in the Evaluator (`EvalCalc`) module as follows:

```
import Values       ( ValType, Name, defaultVal )
import AbSynExpr    ( Expr(..) )
```

```haskell
import Environments ( AnEnv, Name, newEnv, toList, getBinding,
                      hasBinding, newBinding, setBinding,
                      bindList )

type Env = AnEnv ValType
```

42.4.2 Values of AST nodes

We express the *semantics* (i.e. meaning) of the various ELI Calculator language expressions (i.e. nodes of the AST) as follows.

- `c` evaluates to the constant (`NumType`) value `c`.
- `Var n` evaluates to the value of variable `n` in the environment, generating an error if the variable is not defined.
- `Add l r` evaluates to the sum of the values of the expression trees `l` and `r`.
- `Sub l r` evaluates to the difference between the values of the expression trees `l` and `r`.
- `Mul l r` evaluates to the product of the values of the expression trees `l` and `r`.
- `Div l r` evaluates to the quotient of the values of the expression trees `l` and `r`. Division by zero is not defined.

Operations `Add`, `Sub`, `Mul`, and `Div` are *strict*. They are undefined if any of their subexpressions are undefined.

42.4.3 Evaluation function

We can thus define a Haskell *evaluation function* (i.e. *interpreter*) for the ELI Calculator language as follows.

This function in the Evaluator module (`EvalCalc`) does a *post-order traversal* of the abstract syntax tree, first computing the values of the child subexpressions and then computing the value of of a node. The value is returned wrapped in an `Either`, where the `Left` constructor represents an error message and the `Right` constructor a good value.

```haskell
import Values       ( ValType, Name, defaultVal )
import AbSynExpr    ( Expr(..) )
import Environments ( AnEnv, Name, newEnv, toList, getBinding,
                      hasBinding, newBinding, setBinding,
                      bindList )
type EvalErr = String
type Env     = AnEnv ValType
```

```
eval :: Expr -> Env -> Either EvalErr ValType
eval (Val v) _   = Right v
eval (Var n) env =
    case getBinding n env of
        Nothing -> Left ("Undefined variable " ++ n)
        Just i  -> Right i
eval (Add l r) env =
    case (eval l env, eval r env) of
        (Right lv, Right rv) -> Right (lv + rv)
        (Left le,  Left re ) -> Left (le ++ "\n" ++ re)
        (x@(Left le),  _   ) -> x
        (_,     y@(Left le)) -> y
eval (Sub l r) env =
    case (eval l env, eval r env) of
        (Right lv, Right rv) -> Right (lv - rv)
        (Left le,  Left re ) -> Left (le ++ "\n" ++ re)
        (x@(Left le),  _   ) -> x
        (_,     y@(Left le)) -> y
eval (Mul l r) env =
    case (eval l env, eval r env) of
        (Right lv, Right rv) -> Right (lv * rv)
        (Left le,  Left re ) -> Left (le ++ "\n" ++ re)
        (x@(Left le),  _   ) -> x
        (_,     y@(Left le)) -> y
eval (Div l r) env =
    case (eval l env, eval r env) of
        (Right _,  Right 0 ) -> Left "Division by 0"
        (Right lv, Right rv) -> Right (lv `div` rv)
        (Left le,  Left re ) -> Left (le ++ "\n" ++ re)
        (x@(Left le),  _   ) -> x
        (_,     y@(Left le)) -> y
```

Consider an example with a simple `main` function below (that could be added to the `EvalExpr` module) that evaluates the example expressions from a previous section. (See the extended Evaluator module (`EvalCalcExt`).)

```
main =
  do
    let env = [("x",5), ("y",7),("z",1)]
    let exp1 = Val 3          -- 3
    let exp2 = Var "x"        -- x
    let exp3 = Add (Val 1) (Val 2)    -- 1+2
    let exp4 = Add (Var "x") (Val 3)  -- x + 3
    let exp5 = Mul (Add (Var "x") (Var "y"))
                   (Add (Val 2) (Var "z")) -- (x + y) * (2 + z)
    putStrLn ("Expression: " ++ show exp1)
    putStrLn ("Evaluation with x=5, y=7, z=1:  "
                 ++ show (eval exp1 env))
    putStrLn ("Expression: " ++ show exp2)
    putStrLn ("Evaluation with x=5, y=7, z=1:  "
                 ++ show (eval exp2 env))
    putStrLn ("Expression: " ++ show exp3)
    putStrLn ("Evaluation with x=5, y=7, z=1:  "
                 ++ show (eval exp3 env))
    putStrLn ("Expression: " ++ show exp4)
    putStrLn ("Evaluation with x=5, y=7, z=1:  "
                 ++ show (eval exp4 env))
    putStrLn ("Expression: " ++ show exp5)
    putStrLn ("Evaluation with x=5, y=7, z=1:  "
                 ++ show (eval exp5 env))
```

When `main` is called, it first computes he values of the various expressions in the environment `{ x -> 5, y -> 7 }` and then prints their results.

```
Expression: 3
Evaluation with x=5, y=7, z=1:  Right 3
Expression: x
Evaluation with x=5, y=7, z=1:  Right 5
Expression: (+ 1 2)
Evaluation with x=5, y=7, z=1:  Right 3
Expression: (+ x 3)
Evaluation with x=5, y=7, z=1:  Right 8
Expression: (* (+ x y) (+ 2 z))
Evaluation with x=5, y=7, z=1:  Right 36
```

42.5 Simplification

An expression may be more complex than necessary. We can *simplify* it, perhaps with the intention of *optimizing* its evaluation.

An operation whose operands are constants can be simplified by replacing it by the appropriate constant. For example, `Add (Val 3) (Val 4)` is the same semantically as `Val 7`.

Similarly, we can take advantages of an operation's identity element and other mathematical properties to simplify expressions. For example, `Add (Val 0) (Var "x")` is the same as `Var "x"`.

We can thus define a skeletal function `simplify` as follows. As with `eval`, the `simplify` function traverses the abstract syntax tree using a post-order traversal.

```
simplify :: Expr -> Expr
simplify (Add l r) =
    case (simplify l, simplify r) of
        (Val 0, rr)    -> rr
```

```
            (ll, Val 0)    -> ll
            (Val x, Val y) -> Val (x+y)
            (ll, rr)       -> Add ll rr
    simplify (Mul l r) =
        case (simplify l, simplify r) of
            (Val 0, rr)    -> Val 0
            (ll, Val 0)    -> Val 0
            (Val 1, rr)    -> rr
            (ll, Val 1)    -> ll
            (Val x, Val y) -> Val (x*y)
            (ll, rr)       -> Mul ll rr
    simplify t@(Var _) = t
    simplify t@(Val _) = t
```

In an exercise, you are asked to complete the development of this function.

See the incomplete Process AST module (ProcessAST) for the sample code in this section and the next one.

42.6 Symbolic Differentiation

Suppose that we redefine the Expr type to support double precision floating point (i.e. Double) values.

Then let's consider *symbolic differentiation* of the arithmetic expressions. Thinking back to our study of differential calculus, we identify the following rules for differentiation:

- The derivative of a sum is the sum of the derivatives.
- The derivative of a product of two operands is the sum of the product of (a) the first operand and the derivative of the second and (b) the second operand and the derivative of the first.
- The derivative of some variable v is 1 if differentiation is relative to v and is 0 otherwise.
- The derivative of a constant is 0.

We can directly translate these rules into a skeletal Haskell function that uses the above data types, as follows:

```
    deriv :: Expr -> Name -> Expr
    deriv (Add l r) v = Add (deriv l v) (deriv r v)
    deriv (Mul l r) v = Add (Mul l (deriv r v)) (Mul r (deriv l v))
    deriv (Var n)   v
          | v == n    = Val 1
    deriv _         _ = Val 0
```

See the incomplete Process AST module (ProcessAST) for the sample code in this section.

42.7 What Next?

Chapter 41 presented concrete syntax concepts, illustrating them with two different concrete syntaxes for the ELI Calculator language.

This chapter (42) presented abstract syntax trees as structures for representing the essential features of the syntax in a form that can be evaluated directly. The same abstract syntax can encode either of the two concrete syntaxes for the ELI Calculator language.

Chapter 44 introduces lexical analysis and parsing as techniques for processing concrete syntax expressions to generate the equivalent abstract syntax trees.

Before we look at parsing, let's examine the overall modular structure of the ELI Calculator language interpreter in Chapter 43.

42.8 Exercises

1. Extend the abstract syntax tree data type `Expr`, which is defined in the Abstract Syntax module (`AbSynCalc`), to add new operations `Neg` (negation), `Min` (minimum), `Max` (maximum), and `Exp` (exponentiation).

   ```
   data Expr = ...
             | Neg Expr
             | Min Expr Expr
             | Max Expr Expr
             | Exp Expr Expr
             ...
               deriving Show
   ```

   Then extend the `eval` function, which is defined in the Evaluator module (`EvalCalc`), to add these new operations with the following informal semantics:

   - `Neg e` negates the value of expression `e`. For example, `Neg (Val 1)` yields `(Val (-1))`.
   - `Min l r` yields the smaller value of expression `l` and expression `r`.
   - `Max l r` yields the larger value of expression `l` and `r`.
   - `Exp l r` raises the value of expression `l` to a power that is the value of expression `r`. It is undefined for a negative exponent value `r`.

   These operations are all strict; they only have values if all their subexpressions also have values.

2. Extend the `simplify` function to support operations `Sub` and `Div` and the new operations given in the previous exercise.

   This function should simplify the abstract syntax tree by evaluating subexpressions involving only constants (not evaluating variables) and handling special values like identity and zero elements.

3. Extend the `simplify` function from the previous exercise in other ways. For example, take advantage of mathematical properties such as *associativity* (`(x + y) + z = x + (y + z)`), *commutativity* (`x + 1 = 1 + x`), and *idempotence* (`x min x = x`).

4. Extend the abstract syntax tree data type `Expr` to include the binary operators `Eq` (equality) and `Lt` (less-than comparison), logical unary operator `Not`, and the ternary conditional expression `If` (if-then-else).

   ```
   data Expr = ...
             | Eq  Expr Expr
             | Lt  Expr Expr
             | Not Expr
             | If  Expr Expr Expr
             ...
               deriving Show
   ```

   Then extend the `eval` function to implement these new operations.

   This extended language does not have Boolean values. We represent “false” by integer 0 and “true” by a nonzero integer, canonically by 1.

   We can express the informal semantics of the new ELI Calculator language expressions as follows:

   - `Eq l r` yields the value 1 if expressions `l` and `r` have the same value; it yields the value 0 if `l` and `r` have different values.

   - `Lt l r` yields the value 1 if the value of expression `l` is smaller than the value of expression `r`; it yields the value 0 if `l` is greater than or equal to `r`.

   - `Not i` yields 1 if the value of expression `i` is 0; it yields the value 0 if `i` is nonzero.

   - `If c l r` first evaluates expression `c`. If `c` has a nonzero value, the `If` yields the value of expression `l`. If `c` has value 0, the `If` yields the value of expression `r`.

   Operations `Eq`, `Lt`, and `Not` are strict for all subexpressions; that is, they are undefined if any subexpression is undefined.

   Operation `If` is strict in its first subexpression `c`.

   Note: The constants `falseVal` and `trueVal` and the functions `boolToVal` and `valToBool` in the `Values` module may be helpful. (The intention of the

Values module is to keep the representation of the values hidden from the rest of the interpreter. In particular, these constants and functions these are to help encapsulate the representation of booleans as the underlying values.)

5. Extend the abstract syntax tree data type Expr from the previous exercise (which defines operator If) to include a Switch expression.

   ```
   data Expr = ...
             | Switch Expr Expr [Expr]
             ...
               deriving Show
   ```

   Then extend the eval function to implement this new operation.

   We can express the informal semantics of this new ELI Calculator language expression as follows:

   - Switch n def exs first evaluates expression n. If the value of n is greater than or equal to 0 and less than length exs, then the Switch yields the value of the nth expression in list exs (where the first element is at index 0). Otherwise, the Switch yields the value of the default expression def.

6. Develop an object-oriented program (e.g. in Java) to carry out the same functionality as the Expr data type and eval function described in this chapter. That is, define a class hierarchy that corresponds to the Expr data type and use the message-passing style to implement the needed classes and instances.

7. Extend the object-oriented program from the previous exercise to the Neg, Min, Max, and Exp as described in an earlier exercise.

8. Extend the object-oriented program from the previous exercise to implement the Eq, Lt, Not, and If as described in another earlier exercise.

9. Extend the object-oriented program above to implement simplification.

10. For this exercise, redefine the Expr data type above to hold Double constants instead of Int. In addition to Add, Mul, Sub, Div, Neg, Min, Max, and Exp, extend the data type and eval function to include the trigonometric operators Sin and Cos for sine and cosine.

11. Using the extended Double version of Expr from the previous exercise, extend function deriv to support all the operators in the data type.

42.9 Acknowledgements

I initially developed the ELI Calculator language (then called the Expression Language) case study for the Haskell-based offering of CSci 556, Multiparadigm

Programming, in Spring 2017. I based this work, in part, on ideas from:

- the 2016 version of my Scala-based Expression Tree Calculator case study from my *Notes on Scala for Java Programmers* [Cunningham 2018] (which was itself adapted from the the tutorial [Schniz 2018])
- the Lua-based Expression Language 1 and Imperative Core interpreters I developed for the Fall 2016 CSci 450 course
- Kamin's textbook [Kamin 1990] and my work to implement three (Core, Lisp, and Scheme) of these interpeters in Lua in 2013
- sections 1.2, 3.3, and 5.1 of the Linz textbook [Linz 2017]
- section 1.3 of the Sestoft textbook [Sestoft 2012]
- Wikipedia articles [Wikipedia 2018a] on Formal Grammar, Regular Grammar, Context-Free Grammar, Backus-Naur Form, Extended Backus-Naur Form, and Parsing
- the Wikipedia articles [Wikipedia 2018b] on Abstract Syntax and Associative Array.

In 2017, I continued to develop this work as Chapter 10, Expression Language Syntax and Semantics, of my 2017 Haskell-based programming languages textbook.

In Summer 2018, I divided the previous Expression Language Syntax and Semantics chapter into three chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming.* Section 10.2 became Chapter 41, Calculator Concrete Syntax, sections 10.3-5 and 10.7-8 became Chapter 42, Calculator Abstract Syntax & Evaluation (this chapter), and sections 10-6 and 10-9 and section 11.5 were expanded into Chapter 43, Calculator Modular Structure.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

42.10 References

[Cunningham 2018]: H. Conrad Cunningham. *Notes on Scala for Java Programmers*, 2018 (which is itself adapted from the tutorial [Schinz 2018] Scala for Java Programmers

[Kamin 1990]: Samuel N. Kamin. *Programming Languages: An Interpreter-Based Approach*, Addison-Wesley, 1990.

[Linz 2017]: Peter Linz. *An Introduction to Formal Languages and Automata*, Fifth Edition, Jones and Bartlett, 2017.

[Schinz 2018]: Michel Schinz and Philipp Haller. A Scala Tutorial for Java Programmers, Scala Language Website, accessed February 2018.

[Sestoft 2012]: Peter Sestoft. *Programming Language Concepts*, Springer, 2012.

[Wikipedia 2018a]: Wikipedia. Articles on Formal Grammar, Regular Grammar, Context-Free Grammar, Backus-Naur Form, Extended Backus-Naur Form, and Parsing. Accessed 9 August 2018.

[Wikipedia 2018b]: Wikipedia. Articles on Abstract Syntax, Associative Array, Accessed 9 August 2018.

42.11 Terms and Concepts

Abstract syntax, abstract syntax tree (AST), associative data structure, environment, value, semantics, evaluation function, interpreter, simplification, optimization, symbolic differentiation, associativity, commutativity (symmetry), idempotence.