Exploring Languages
with Interpreters
and Functional Programming
Chapter 43

H. Conrad Cunningham

27 April 2022

Contents

43 Calculator: Modular Structure **2**

Professor of Computer and Information Science
University of Mississippi
214 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-7396 (dept. office)

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of April 2022 is a recent version of Firefox from Mozilla.

43 Calculator: Modular Structure

43.1 Chapter Introduction

TODO: Write missing pieces and flesh out other sections

43.2 Module Dependencies

An ELI Calculator interpreter consists of seven modules. The dependencies among modules as shown in Figure 43.1. (The module at the tail of an arrow depends on the module at the head.)

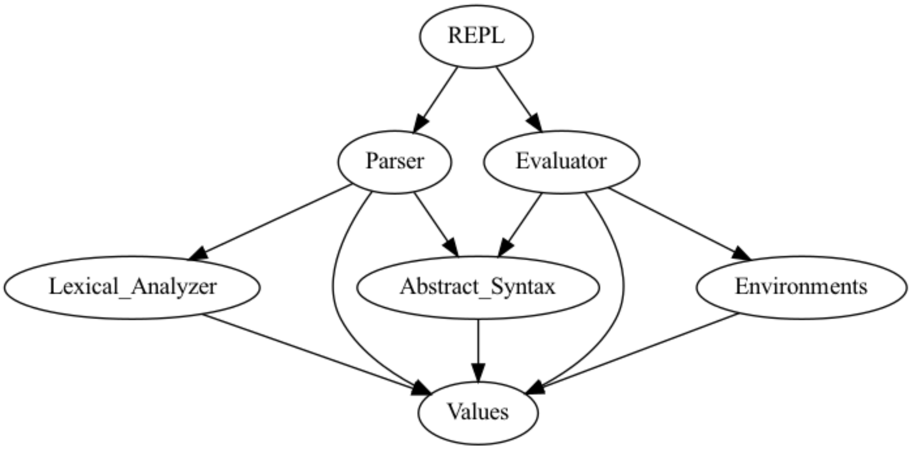

Figure 43.1: ELI Calculator language module dependencies.

We examine each module in the following sections.

TODO: Some of these are concrete modules intended for direct use by all implementations. Some are concrete modules intended for use by just ELI Calculator. Some are "abstract modules" intended to define an interface for implementation by each language as needed. Some may, in some sense, define a module role (e.g., same secret) that must be satisfied for all languages, but which may have a different abstract interface. Etc. This probably should be clarified for each module after study and thought.

43.3 Values Module

The *Values* module `Values` was introduced in Chapter 42. It encapsulates the definitions and functions that know the specific representation of an ELI language's data. Other modules for that language should use its public features to enable the representation to be changed easily.

The *secret* of the information-hiding module Values is the specific representation for the values supported by the language.

This module currently supports both the ELI Calculator language and the ELI Imperative Core language we examine in later chapters. For both languages, the only type of values supported are integers. Booleans are encoded as integers.

The Values module's *abstract interface* includes the following public features:

- Type `ValType` is the type of the values in the ELI language.
- Constant `defaultVal` is the default value for ELI language variables when no value is specified.

  Note: A *constant* is an argumentless function in Haskell.
- Constants `falseVal` and `trueVal` are the ELI language's canonical representations for false and true as `ValType` values, respectively.
- Function `boolToVal` converts Haskell `Bool` values `False` and `True` to `falseVal` and `trueVal`, respectively.
- Function `valToBool v` converts ELI language value `v` to Haskell `False` and `True` appropriately.

  `falseVal` is mapped to Haskell `False`. Any other value is mapped to Haskell `True`; we call these *truthy* values.

If a language supports types other than integers, then that language will need a variant of the Values module that redefines `ValType` accordingly and perhaps defines additional public functions. However, the redefined module should seek to preserve the secret and other features of the abstract interface.

The interface also includes the following, which are intended for the exclusive use of the lexical analysis module to support finite range integers (e.g., a string representation of an integer that is beyond the range of `Int`).

- Type `NumType` is the actual type used to represent integers.
- Function `toNumType` takes a string of digits `numstr` and returns an `Either String NumType` where `Left` wraps an error message and `Right` wraps `numstr` interpreted as a `NumType` value.

TODO: Review how integer constant overflow is handled and seek to encapsulate the representation better. Also might comment that the knowledge of the value representation is probably shared between the Values and Lexical Analysis modules.

The Values module does not depend upon any other modules. All other current modules depend upon it directly except the user-interface module REPL.

43.4 Environments Module

An *environment* is a mapping between a name and its value.

The *Environments* module `Environments` was introduced in Chapter 42. It encapsulates the definitions and functions that know the specific representation of an environment for an ELI language. Other modules should use its public features to enable the representation to be changed easily.

The *secret* of the information-hiding module Environments is the specific representation for the environments used by the language's interpreter. This module currently supports both the ELI Calculator and the ELI Imperative Core languages (defined in future chapters). Given that the "value" is a polymorphic parameter, it should work for most languages unless the nature of names changes significantly.

- The ELI Calculator language creates a single global environment consisting of a set of `(Name,ValType)` pairs that map variables to their values.
- The ELI Imperative Core language (which also supports function definitions and function calls) creartes three different environments, all of which are implemented with the Environments module:
    - a global variable environment consisting of a set of `(Name,ValType)` pairs (as above)
    - a global function definition environment consisting of a set of '`Name`-function definition pairs
    - a local parameter environment like the global variable environment except holding the values of the parameters for a function call

The Environments module's *abstract interface* includes the following public features.

- Type `AnEnv a` is the type of an environment whose values have polymorphic parameter type `a`.
- Type `Name` is imported from the `Values` module and reexported.
- Constructor function `newEnv` returns a new empty environment.
- Mutator function `newBinding` adds a new name-value binding to an environment.
- Mutator function `setBinding` changes the value of an existing name in an environment.
- Mutator function `bindList` takes a list of name-value pairs and adds a new binding for each to an environment.
- Accessor function `toList` returns an association list equivalent to the environment.
- Accessor function `getBinding` returns the value associated with a given name.

- Query function `hasBinding` returns `True` if and only if the given name is bound in the environment.

The Environments module depends upon the Values module and the Evaluator module depends upon it.

43.5 Abstract Syntax Module

The *Abstract Synax* module `AbSynCalc` module was introduced in Chapter 42. It centralizes the abstract syntax definition for the ELI Calculator language so it can be imported where needed.

The abstract syntax consists of algebraic data type definitions. The semantics of the abstract syntax tree is known by modules that must create (e.g., parser) and use (e.g., evaluator) the abstract syntax trees.

TODO: Review how the AST semantics is handled to see if it can be better encapsulated. But remember that too much abstraction may make the pedagogical goals more difficult to achieve (e.g., exercises to add new elements to the abstract syntax and semantics).

The ELI Calculator Language's Abstract Syntax module defines and exports the algebraic data type `Expr` and implements it as an instance of class `Show`. Values of type `Expr` are the abstract syntax trees for the ELI Calculator language.

The module also exports types `ValType` and `Name` that it imports from the Values module.

The equivalent modules for other languages must define the abstract syntax for that language using appropriate algebraic data types that are instances of `Show`. They should, however, use

The Abstract Syntax module depends upon the Values module and the Evaluator and Parser modules depend upon it.

43.6 Evaluator Module

The *Evaluator* module `EvalCalc` was introduced in Chapter 42. It encapsulates the definition of the evaluation function (i.e., the semantics) of the ELI Calculator language.

TODO: Consider how to handle the extensions to the Evaluator module in Chapter 42 for simplification and differentiation (i.e., `ProcessAST` module).

The *secret* of the `EvalCalc` is the implementation of the semantics of the language, including the specifics of the environment. Currently, some aspects of the language semantics are not completely encapsulated within the Evaluator module; they are shared with the Parser module (which creates the abstract syntax trees initially).

TODO: Explore whether the semantics can be better encapsulated and continue to meet the pedagogical goals of the interpreter.

The Evaluator module's *abstract interface* includes the following public features.

TODO: Perhaps simply call this an "interface" because it is not likely used by more than one concrete implementation.

- Evaluation function `eval` takes an ELI Calculator abstract syntax tree (i.e., an `Expr`) and returns its value in the environment.
- Type `Env` defines the environment (i.e., mapping of variable names to their values) for the ELI Calculator language.
- Constant `lastVal` is the variable name whose value in the environment is the result of the most recent expression evaluation.
- Constructor function `newEnviron` creates a new environment that is empty except that variable `lastVal` is set to `Values.defaultVal`.
- Query function `hasNameBinding` returns `True` if and only if the given name is defined in the environment.
- Mutator function `newNameBinding` that creates a new variable in the environment and gives it a value.
- Mutator function `setNameBinding` that sets an existing variable in the environment to a new value.
- Accessor function `getNameBinding` retrieves the value of a variable from the environment.
- Accessor function `showEnviron` displays all the variables and their values in the environment.
- Type `EvalErr` represents error messages arising from evaluation.
- Types `ValType` and `Name` are imported from the Values module and reexported.
- Type `Expr` is imported from the Abstract Syntax module and reexported.

TODO: Comment on how the above secret should be preserved and might need to be modified for other ELI languages.

The Evaluator module depends directly upon the Abstract Syntax, Environments, and Values modules. The language's user-interface module REPL depends upon it. However, as noted above, the Evaluator and Parser modules currently share some aspects of the language semantics.

43.7 Lexical Analysis Module

The *Lexical Analyzer* module `LexCalc` is introduced in Chapter 44. It is common to both the prefix and infix parsers for the ELI Calculator language.

The *secret* of this module is the lexical structure of the concrete language syntax.

The Lexical Analyzer module's *abstract interface* consists of the following public features.

- Algebraic data type `Token` describes the smallest units of the syntax processed by the parser, such as identifiers, operator symbols, parentheses, etc.
- Function `showTokens` is a convenience function that shows a list of tokens as a string.
- Function `lexx` takes a string and returns the corresponding list of lexical tokens, but it does not distinguish among identifiers, keywords, and operators.
- Function `lexer` takes a string and returns the corresponding list of lexical tokens, distinguishing among identifiers, keywords, and operators.
- Type `NumType` is imported from the Values module and reexported; it is the actual type used to represent integers.
- Type `Name`{.haskell is from the Values module and reexported; it is the type that represents "names" such as identifiers and operator symbols.

TODO: Consider whether the above should just be an interface rather than an abstract interface. Also how should the secret and interface be preserved and modified for other languages. Also consider what I should say below about the special dependence upon the Values module and any sharing of information about values.

The Lexical Analyzer module depends upon the Values module and the Parser module depends upon it.

43.8 Parser Modules

Chapter 44 introduces two alternative implementations of the *Parser* abstract module for the ELI Calculator language. These implementations correspond to the two different concrete syntaxes given in Chapter 41. Both use the same Lexical Analyzer.

- Module `ParsePrefixCalc` parses an ELI Calculator language *prefix* expression and generates the equivalent abstract syntax tree.
- Module `ParseInfixCalc` parses an ELI Calculator language *infix* expression and generates the equivalent abstract syntax tree,

The *secret* of the abstract parser module is how the input syntax is recognized and translated to the abstract syntax.

The Parser abstract module's *abstract interface* consists of the following public features.

- Function `parse` takes an input string, parses it according to the corresponding ELI Calculator language concrete syntax and returns an `Either` item wrapping the `Expr` abstract syntax tree (`Right`) or an error message (`Left`).
- Function `parseExpression` takes a `Token` list, parses an `Expr` from the beginning of the list, and returns a pair consisting of
  - an `Either` wrapping the `Expr` abstract syntax tree found (`Right` or an error message (`Right`
  - the `Token` list remaining after the `Expr`.
- Type `ParErr` is the type of the error messages.
- Function `trimComment` trims an end-of-line comment from a line of text.
- Function `getName` takes a string and returns a `Just` wrapping a `Name` if it is a valid identifier or a `Nothing` if any non-identifier characters occur.
- Function `getValue` extracts an identifier from the beginning of a string and returns the identifier and the remaining string.
- Types `ValType` and `Name` are imported from the Values module and reexported.
- Type `Expr` is imported from the Abstract Syntax module and reexported.

TODO: Comment on how the above secret should be preserved and might need to be modified for other ELI languages.

The Parser module depends directly upon the Lexical Analyzer, Abstract Syntax, and Values modules. The language's user-interface module REPL depends upon it. However, as noted above, the Evaluator and Parser modules currently share some aspects of the language semantics.

43.9 REPL Modules

A REPL (Read-Evaluate-Print Loop) is a command line user interface with the following cycle of steps:

1. *Read* an input from the command line.

   If the input is an exit command, exitloop ; else continue.
2. *Evaluate* the expression after parsing.
3. *Print* the resulting value.
4. *Loop* back to step 1.

The *secret* of the REPL modules is how the user interacts with the interpreter.

The ELI Calculator language interpreter provides two REPL modules:

- `PrefixCalcREPL` that uses the Calculator language's prefix syntax
- `InfixCalcREPL` that uses the Calculator languages's infix syntax

In addition to accepting ELI Calculator expressions, they accept the REPL commands `:set`, `:display`, and `:quit`.

TODO: What about `:use`? Do I need to elaborate on the commands further? Probably.

TODO: The REPL functions need to be refactored. Also the issue of the `:use` command versus a `use` expression in the language needs to be reconsidered.

The REPL module depends directly upon the Parser and Evaluator modules. No other modules depend upon it.

43.10 Code Improvement Modules

TODO: Consider how this should be presented in both Chapter 42 and 43.

In addition, the partially implemented *Process AST* module includes the skeleton `simplify` and `deriv` functions discussed in Chapter 42.

This module is "wrapper" for the `EvalCalc` module currently.

43.11 What Next?

TODO

43.12 Chapter Source Code

The ELI Calculator language interpreter includes the following source code modules:

- *Values* module `Values`
- *Environments* module `Environments`
- *Abstract Synax* module `AbSynCalc`
- *Evaluator* module `EvalCalc`
- *Lexical Analyzer* module `LexCalc`
- *Parser* modules
  - Prefix parser `ParsePrefixCalc`
  - Infix parser `ParseInfixCalc`
- *REPL* modules
  - Prefix REPL `PrefixCalcREPL`
  - Infix REPL `InfixCalcREPL`

- Skeleton simplify and derivative module `ProcessAST`

43.13 Exercises

TODO

43.14 Acknowledgements

For the general acknowledgements for the ELI Calculator case study and Chapters 41-46 through Spring 2019, see the Acknowledgements section of Chapter 41.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a unified bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

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.

43.15 Terms and Concepts

TODO

43.16 References