Exploring Languages with Interpreters and Functional Programming Chapter 43

H. Conrad Cunningham

24 November 2018

Contents

43 Calculator: Modular Structure **2**
- 43.1 Chapter Introduction 2
- 43.2 Module Dependencies 2
- 43.3 Values Module 2
- 43.4 Environments Module 3
- 43.5 Abstract Syntax Module 4
- 43.6 Evaluator Module 5
- 43.7 Lexical Analysis Module 5
- 43.8 Parser Modules 6
- 43.9 REPL Modules 7
- 43.10 Code Improvement Modules 7
- 43.11 What Next? 7
- 43.12 Exercises 8
- 43.13 Acknowledgements 8
- 43.14 References 9
- 43.15 Terms and Concepts 9

Copyright (C) 2017, 2018, H. Conrad Cunningham
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.

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 with the module dependencies shown in Figure 43-1.

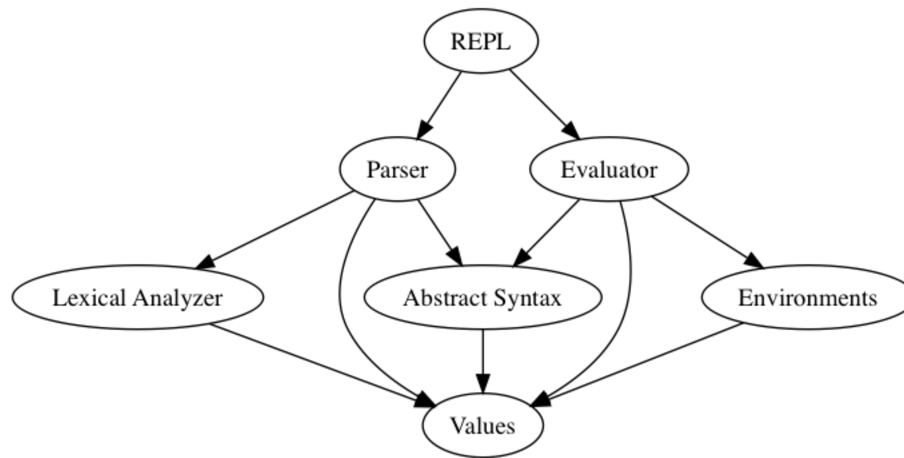

Figure 43-1: ELI Calculator language module dependencies

We examine each module in the following sections.

43.3 Values Module

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

The *secret* of the Values module 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 a later chapter. 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.
- 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.

The interface also includes the following, which are intended for the exclusive use of the lexical analysis module to support finite range integers.

- 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.

43.4 Environments Module

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

The *Environments* module `Environments` is 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 Environments module is the specific representation for the environments used in interpreter for the ELI language.

This module currently supports both the ELI Calculator and the ELI Imperative Core languages (in a future chapter).

- The ELI Calculator language uses a single global environment consisting of a set of `(Name,ValType)` pairs.
- The ELI Imperative Core language (which supports function definitions and function calls) uses 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 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.

43.5 Abstract Syntax Module

The *Abstract_Synax* module `AbSynCalc` module is 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.

The 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 the Values module.

43.6 Evaluator Module

The *Evaluator* module `EvalCalc` encapsulates the definition of the evaluation function (i.e. the semantics) of the ELI Calculator language.

The *secret* of the `EvalCalc` is the implementation of the semantics of the language, including the specifics of the environment.

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

- 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.

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.

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.

43.9 REPL Modules

A REPL (Read-Evaluate-Print Loop) is a command line 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`.

43.10 Code Improvement Modules

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 Exercises

TODO

43.13 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 and 1.4 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 42, Calculator Concrete Syntax, sections 10.3-5 and 10.7-8 became chapter 43, Calculator Abstract Syntax & Evaluation, and sections 10-6 and 10-9 and section 11.5 became Chapter 44, Calculator Architecture (this chapter).

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, and sections 10-6 and 10-9 and section 11.5 were expanded into Chapter 43, Calculator Modular Structure (this chapter).

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.

43.14 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 and Associative Array, Accessed 9 August 2018.

43.15 Terms and Concepts

TODO