Exploring Languages with Interpreters and Functional Programming

Chapter 44

H. Conrad Cunningham

16 August 2018

Contents

44 Calculator: Modular Structure 2
44.1 Chapter Introduction 2
44.2 Module Dependencies 2
44.3 Module Descriptions 2
44.3.1 Values module 2
44.3.2 Environments module 3
44.3.3 Abstract syntax module 4
44.3.4 Evaluator module 5
44.3.5 Lexical analysis module 5
44.3.6 Parser modules 6
44.3.7 Simplification and Derivative modules 6
44.4 Read-Evaluate-Print Loop (REPL) 6
44.5 What Next? 6
44.6 Exercises 6
44.7 Acknowledgements 7
44.8 References 7
44.9 Terms and Concepts 8

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 August 2018 is a recent version of Firefox from Mozilla.
44 Calculator: Modular Structure

44.1 Chapter Introduction

TODO

44.2 Module Dependencies

An ELI Calculator interpreter consists of seven modules with the module dependencies shown in Figure 44-1.

![Module Dependencies Diagram]

Figure 44-1: ELI Calculator Language module dependencies

We examine each module in the following.

44.3 Module Descriptions

44.3.1 Values module

The Values module 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`, the type of the values in the ELI language
- constant `defaultVal`, the default value for ELI language variables when no value is specified

Note: A constant is an argumentless function.

- constants `falseVal` and `trueVal`, the ELI language's canonical representations for false and true as `ValType` values, respectively
- function `boolToVal` that converts Haskell `Bool` values `False` and `True` to `falseVal` and `trueVal`, respectively
- function `valToBool v` that 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`, the actual type used to represent integers
- function `toNumType` that 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

### 44.3.2 Environments module

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

The *Environments* module *Environments* 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.

- 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 \((\text{Name}, \text{ValType})\) pairs (as above)
– a global function definition environment consisting of a set of ‘\text{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 \text{AnEnv} a, the type of an environment whose values have parameter type \text{a}
- type \text{Name} imported from the \text{Values} module
- constructor function \text{newEnv} that returns a new empty environment
- mutator function \text{newBinding}, which adds a new name-value binding to an environment
- mutator function \text{setBinding}, which changes the value of an existing name in an environment
- mutator function \text{bindList} that takes a list of name-value pairs and adds a new binding for each to an environment
- accessor function \text{toList} that returns an association list equivalent to the environment
- accessor function \text{getBinding} that returns the value associated with a given name
- query function \text{hasBinding} that returns \text{True} if and only if the given name is bound in the environment

44.3.3 Abstract syntax module

The \text{Abstract_Synax} module \text{AbSynCalc} module centralizes the abstract syntax definition for the ELI Calculator language so it can be imported where needed.

The module defines and exports the algebraic data type \text{Expr} and implements it as an instance of class \text{Show}. Values of type \text{Expr} are abstract syntax trees for the ELI Calculator language.

The module also exports types \text{ValType} and \text{Name} that it imports from the \text{Values} module.
44.3.4 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` that takes an ELI Calculator abstract syntax tree (i.e. an `Expr`) and returns its value in the environment
- type `Env` that defines the environment (i.e. mapping of variable names to their values) for the ELI Calculator language
- constant `lastVal`, the variable name whose value in the environment is the result of the most recent expression
- constructor function `newEnviron` that creates a new environment that is empty except that variable `lastVal` is set to `Values.defaultVal`
- query function `hasNameBinding` that 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` that retrieves the value of a variable from the environment
- accessor function `showEnviron` that displays all the variables and their values in the environment
- type `EvalErr` for error messages
- types `ValType` and `Name` imported from the Values module
- type `Expr` imported from the Abstract Syntax module

44.3.5 Lexical analysis module

TODO

The *Lexical Analyzer* module `LexCalc` is common to both the prefix and infix parsers for the ELI Calculator language.
44.3.6 Parser modules

TODO
- Prefix syntax
  - Recursive descent Parser module for prefix language ParsePrefixCalc
  - REPL module for prefix syntax PrefixCalcREPL
- Infix syntax
  - Recursive descent Parser module for infix language ParseInfixCalc
  - REPL module for infix syntax InfixCalcREPL

44.3.7 Simplification and Derivative modules

TODO
In addition, the code for the partially implemented Process AST module includes the skeleton simplify and deriv code:
- Incomplete Process AST module ProcessAST (simplify and deriv)
This module is “wrapper” for the EvalCalc module currently.

44.4 Read-Evaluate-Print Loop (REPL)

TODO: Write this section
- Read expression from command line
- Evaluate expression after parsing
- Print resulting value
- Loop back for next expression

44.5 What Next?

TODO

44.6 Exercises

TODO
44.7 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 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 interpreters 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]

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).

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.

44.8 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
44.9 Terms and Concepts

TODO