Exploring Languages with Interpreters and Functional Programming Chapter 41

H. Conrad Cunningham

04 April 2022

Contents

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.

41 Calculator: Concrete Syntax

41.1 Chapter Introduction

Chapter 40 surveyed the overall language processing pipeline.

Beginning with this chapter, we explore language concepts processing techniques in the context of a simple case study. The case study uses a language of simple arithmetic expressions, a language we call the *ELI (Exploring Languages with Interpreters) Calculator language*.

- Chapter 41 introduces the formal concepts related to concrete syntax. It gives two different concrete syntaxes for the ELI Calculator language.
- Chapter 42 introduces the concepts of abstract syntax and language semantics. It represents both concrete syntaxes of the ELI Calculator language with the same abstract syntax encoded as a Haskell algebraic data type. It defines the semantics of the language using a Haskell function that evaluates (i.e., interprets) the abstract syntax expressions.
- Chapter 43 surveys the modular design and implementation of the ELI Calculator language application.
- Chapter 44 considers lexical analysis and parsing of the concrete syntaxes to generate the corresponding abstract syntax trees
- Chapter 45 explores the construction of a set of parsing combinators.
- Chapter 46 looks at a simple Stack Virtual Machine with an instruction set represented as another algebraic data type and how to translate (i.e., compile), how to execute the machine, and how to translate the abstract syntax trees to sequences of instructions.

We will extend the language with other features in later chapters.

TODO: Give chapter's goals explicitly.

The goals of this chapter are to:

- TODO

41.2 Concrete Syntax

The ELI Calculator language can be represented as human-readable text strings in forms similar to traditional mathematical and programming notations. The structure of these textual expressions is called the *concrete syntax* [20] of the expressions.

In this case study, we examine two possible concrete syntaxes: a familiar infix syntax and a (probably less familiar) parenthesized prefix syntax.

But, first, let's consider how we can describe the syntax of a language.

41.3 Grammars

We usually describe the syntax of a language using a *formal grammar* [7,11].

Formally, a formal grammar consists of a tuple (V, T, S, P), where:

- V is a finite set of *variable* (or *nonterminal*) symbols
- T is a finite set of *terminal* symbols (called the *alphabet*)
- $S \in V$ is the *start* (or *goal*) symbol
- P is a finite set of *production* rules
- V and T are disjoint

Production rules describe how the grammar transforms one sequence of symbols to another. The rules have the general form

$$x \to y$$

where x and y are sequences of symbols from $V \cup T$ such that x has length of at least one symbol.

A *sentence* in a language consists of any finite sequence of symbols that can be generated from the start symbol of a grammar by a finite sequence of productions from the grammar.

We call a sequence of productions that generates a sentence a *derivation* for that sentence.

Any intermediate sequence of symbols in a derivation is called a *sentential form*.

The *language* generated by the grammar is the set of all sentences that can be generated by the grammar.

41.3.1 Context-free grammars and BNF

To express the syntax of programming languages, we normally restrict ourselves to the family of *context-free grammars* (and its subfamilies) [7,11,12] context free. In a context-free grammar (CFG), the production rules have the form

$$A \to y$$

where $A \in V$ and y is a sequence of zero or more symbols from $V \cup T$. This means that an occurence of nonterminal A can be replaced by the sequence x.

We often express a grammar using a metalanguage such as the *Backus-Naur Form (BNF)* or *extended Backus-Naur Form (BNF)* [5,13,14].

For example, consider the following BNF description of a grammar for the unsigned binary integers:

```
<binary> ::= <digit>
<binary> ::= <digit> <binary>
<digit>  ::= '0'
<digit>  ::= '1'
```

The nonterminals are the symbols shown in angle brackets: `<binary>` and `<digit>`.

The terminals are the symbols shown in single quotes: `'0'` and `'1'`.

The production rules are shown with a nonterminal on the left side of the metasymbol `::=` and its replacement sequence of nonterminal and terminal symbols on the right side.

Unless otherwise noted, the start symbol is the nonterminal on the left side of the first production rule.

For multiple rules with the same left side, we can use the `|` metasymbol to write the alternative right sides concisely. The four rules above can be written as follows:

```
<binary> ::= <digit> | <digit> <binary>
<digit>  ::= '0' | '1'
```

We can also use the extended BNF metasymbols:

- `{` and `}` to denote that the symbols between the braces are repeated zero or more times
- `[` and `]` to denote that the symbols between the brackets are optional (i.e., occur at most once)

41.3.2 Derivations

Consider a derivation of the sentence `101` using the grammar for unsigned binary numbers above.

- Start symbol — `<binary>`
- Apply rule 2 — `<digit> <binary>`
- Apply rule 2 — `<digit> <digit> <binary>`
- Apply rule 3 — `<digit> 0 <binary>`
- Apply rule 4 — `1 0 <binary>`
- Apply rule 1 — `1 0 <digit>`
- Apply rule 4 — `1 0 1`

This is not the only possible derivation for `101`. Let's consider a second derivation of `101`.

- Start symbol — `<binary>`
- Apply rule 2 — `<digit> <binary>`
- Apply rule 4 — `1 <binary>`
- Apply rule 2 — `1 <digit> <binary>`
- Apply rule 3 — `1 0 <binary>`
- Apply rule 1 — `1 0 <digit>`
- Apply rule 4 — `1 0 1`

The second derivation applies the same rules the same number of times, but it applies them in a different order. This case is called the *leftmost derivation* because it always replaces the leftmost nonterminal in the sentential form.

Both of the above derivations can be represented by the *derivation tree* (or *parse tree*) [7,20] shown in Figure 41.1. (The numbers below the nodes show the rules applied.)

41.3.3 Regular grammars

The grammar above for binary numbers is a special case of a context-free grammar called a *right-linear grammar* [7,15]. In a right-linear grammar, *all* productions are of the forms

$A \rightarrow xB$
$A \rightarrow x$

where A and B are nonterminals and x is a sequence of zero or more terminals. Similarly, a *left-linear grammar* [7,15] must have *all* productions of the form:

$A \rightarrow Bx$
$A \rightarrow x$

A grammar that is either right-linear or left-linear is called a *regular grammar* [7,16].

(Note that *all* productions in a grammar must satisfy either the right- or left-linear definitions. They cannot be mixed.)

We can recognize sentences in a regular grammar with a simple "machine" (program)—a *deterministic finite automaton (DFA)* [7,18].

In general, we must use a more complex "machine"—a *pushdown automaton (PDA)*[7,19]—to recognize a context-free grammar.

We leave a more detailed study of regular and context-free grammars to courses on formal languages, automata, or compiler construction.

Now let's consider the concrete syntaxes for the ELI Calculator language—first infix, then prefix.

41.4 Infix syntax

An *infix syntax* for expressions is a syntax in which most binary operators appear between their operands as we tend to write them in mathematics and in programming languages such as Java and Haskell. For example, the following are intended to be valid infix expressions:

```
3
-3
x
```

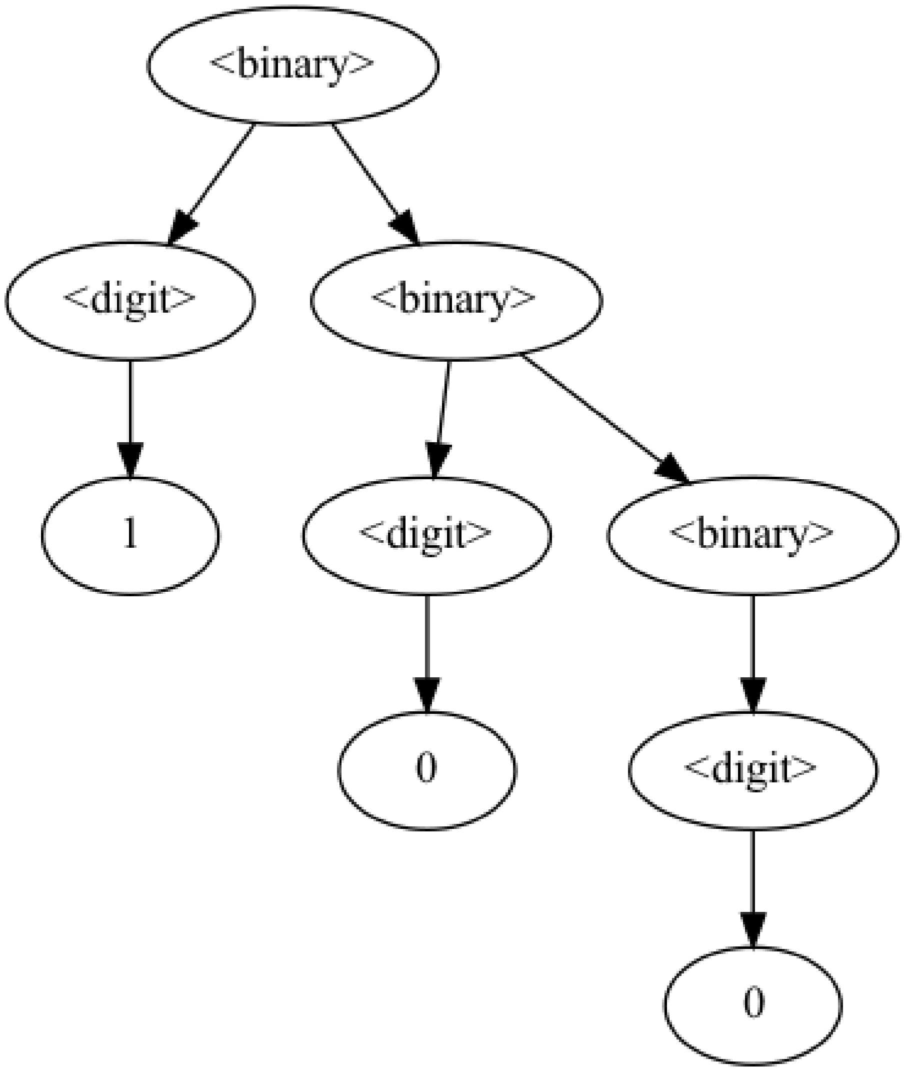

Figure 41.1: Derivation (parse) tree for binary number 101.

```
1+1
x + 3
(x + y) * (2 + z)
```

For example, we can present the concrete syntax of our core Calculator language with the grammar below. Here we just consider expressions made up of decimal integer constants; variable names; binary operators for addition, subtraction, multiplication, and division; and parentheses to delimit nested expressions.

We express the upper levels of the infix expression's syntax with the following context-free grammar where `<expression>` is the start symbol.

```
<expression> ::= <term> { <addop> <term> }
<term>       ::= <factor> { <mulop> <factor> }
<factor>     ::= <var> | <val>
               | '(' <expression> ')'
<val>        ::= [ '-' ] <unsigned>
<var>        ::= <id>
<addop>      ::=  '+'  |  '-'
<mulop>      ::=  '*'  |  '/'
```

Normally we want operators such as multiplication and division to bind more tightly than addition and subtraction. That is, we want expression `x + y * z` to have the same meaning as `x + (y * z)`. To accomplish this in the context-free grammar, we position `<addop>` in a higher-level grammar rule than `<mulop>`.

We can express the lower (lexical) level of the expression's grammar with the following production rules:

```
<id>         ::=  <firstid>  |  <firstid> <idseq>
<idseq>      ::=  <restid>   |  <restid> <idseq>
<firstid>    ::=  <alpha>    | '_'
<restid>     ::=  <alpha>    | '_'  | <digit>
<unsigned>   ::=  <digit>    |  <digit> <unsigned>
<digit>      ::=  any numeric character
<alpha>      ::=  any alphabetic character
```

The variables `<digit>` and `<alpha>` are essentially terminals. Thus the above is a regular grammar. (We can also add the rules for recognition of `<addop>` and `<mulop>` and rules for recognition of the terminals `(`, `)`, and `-` to the regular grammar.)

We assume that identifiers and constants extend as far to the "right" as possible. That is, an `<id>` begins with an alphabetic or underscore character and extends until it is terminated by some character other than an alphabetic, numeric, or underscore character (e.g., by whitespace or special character). Similarly for `<unsigned>`.

Otherwise, the language grammar ignores whitespace characters (e.g., blanks, tabs, and newlines). The language also supports end of line comments, any characters on a line following a `--` (double dash).

We can use a *parsing* program (i.e., a *parser*) to determine whether a concrete expression (e.g., `1 + 1`) satisfies the grammar and to build a corresponding *parse tree* [7,21].

Aside: In a previous section, we use the term derivation tree to refer to a tree that we construct from the root toward the leaves by applying production rules from the grammar. We usually call the same tree a parse tree if we construct it from the leaves (a sentence) toward the root.

Figure 41.2 shows the parse tree for infix expression `1 + 1`. It has `<expression>` at its root. The children of a node in the parse tree depend upon the grammar rule application needed to generate the concrete expression. Thus the root `<expression>` has either one child—a `<term>` subtree—or three children—a `<term>` subtree, an `<addop>` subtree, and an `<expression>` subtree.

If the parsing program returns a boolean result instead of building a parse tree, we sometimes call it a *recognizer* program.

41.5 Prefix syntax

An alternative is to use a *parenthesized prefix syntax* for the expressions. This is a syntax in which expressions involving operators are of the form

```
( op operands )
```

where `op` denotes some "operator" and `operands` denotes a sequence of zero or more expressions that are the arguments of the given operator. This is a syntax similar to the language Lisp.

In this syntax, the examples from the section on the infix syntax can be expressed something like:

```
3
3
x
(+ 1 1)
(+ x 3)
(* (+ x y) (+ 2 z))
```

We express the upper levels of a prefix expression's syntax with the following context-free grammar, where `<expression>` is the start symbol.

```
<expression> ::=  <var> | <val> | <operexpr>
<var>        ::=  <id>
<val>        ::=  [ "-" ] <unsigned>
<operexpr>   ::=  '(' <operator> <operandseq> ')'
<operandseq> ::=  { <expression> }
<operator>   ::=  '+'  |  '*'  |  '-'  |  '/' | ...
```

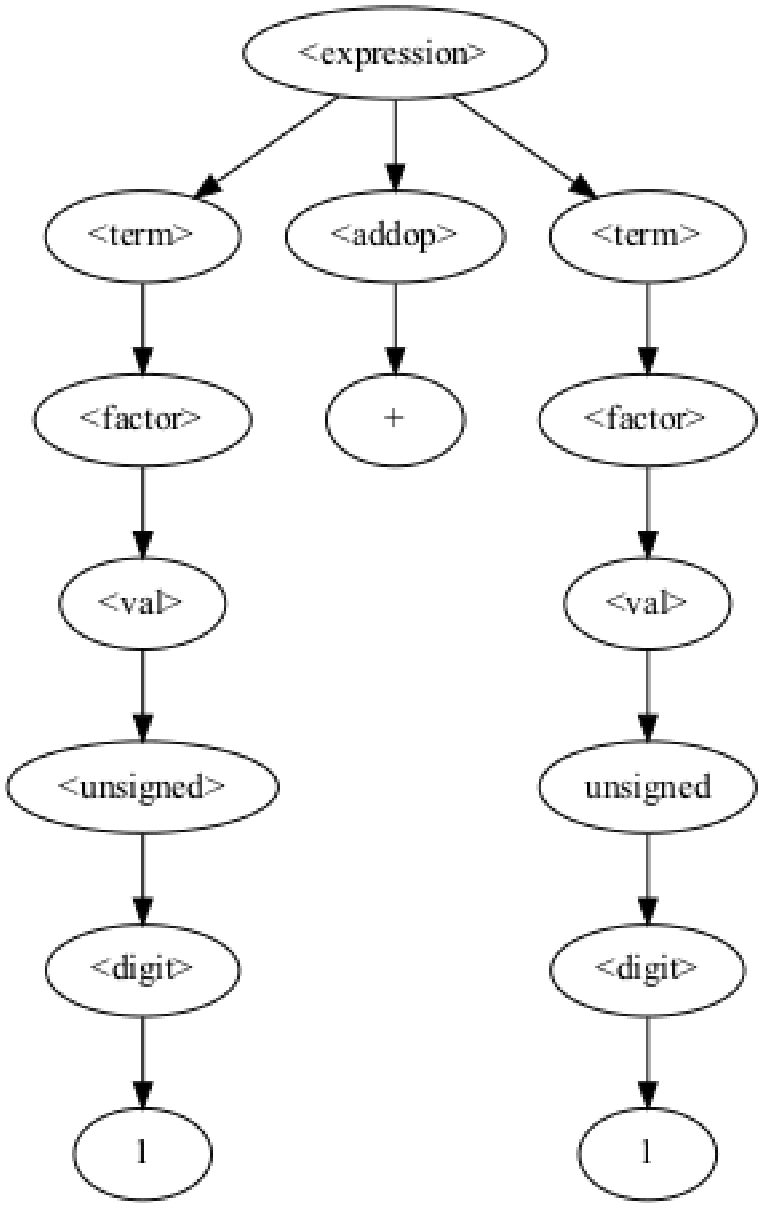

Figure 41.2: Parse tree for infix 1 + 1.

We can express the lower (lexical) level of the expression's grammar with basically the same regular grammar as with the infix syntax. (We can also add the rule for recognition of `<operator>` and for recognition of the terminals `(`, `)`, and `-` to the regular grammar

The parse tree for prefix expression `(+ 1 1)` is shown in Figure 41.3,

Because the prefix syntax expresses all operations in a fully parenthesized form, there is no need to consider the binding powers of operators. This makes parsing easier.

The prefix also makes extending the language to other operators—and keywords—much easier. Thus we will primarily use the prefix syntax in this and other cases studies.

We return to the problem of parsing expressions in a later chapter.

41.6 What Next?

This chapter (41) introduced the formal concepts related to a language's concrete syntax. It also introduced the *ELI (Exploring Languages with Interpreters) Calculator language*, which is the simple language we use in the following five chapters.

Chapter 42 examines the concepts of abstract syntax and evaluation, using the ELI Calculator language as an example.

41.7 Chapter Source Code

TODO if needed

41.8 Exercises

TODO

41.9 Acknowledgements

Chapters 41-46 of this book explore the ELI Calculator language and general concepts and techniques for language processing. 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 continued this work during Summer and Fall 2017 for the Fall 2017 offering of CSci 450, Organization of Programming Languages. I based the ELI Calculator language case study on ideas drawn, in part, from the following:

- the 2016 version of my Scala-based Expression Tree Calculator case study from my *Notes on Scala for Java Programmers* [4] (which was itself adapted from the the tutorial [8])

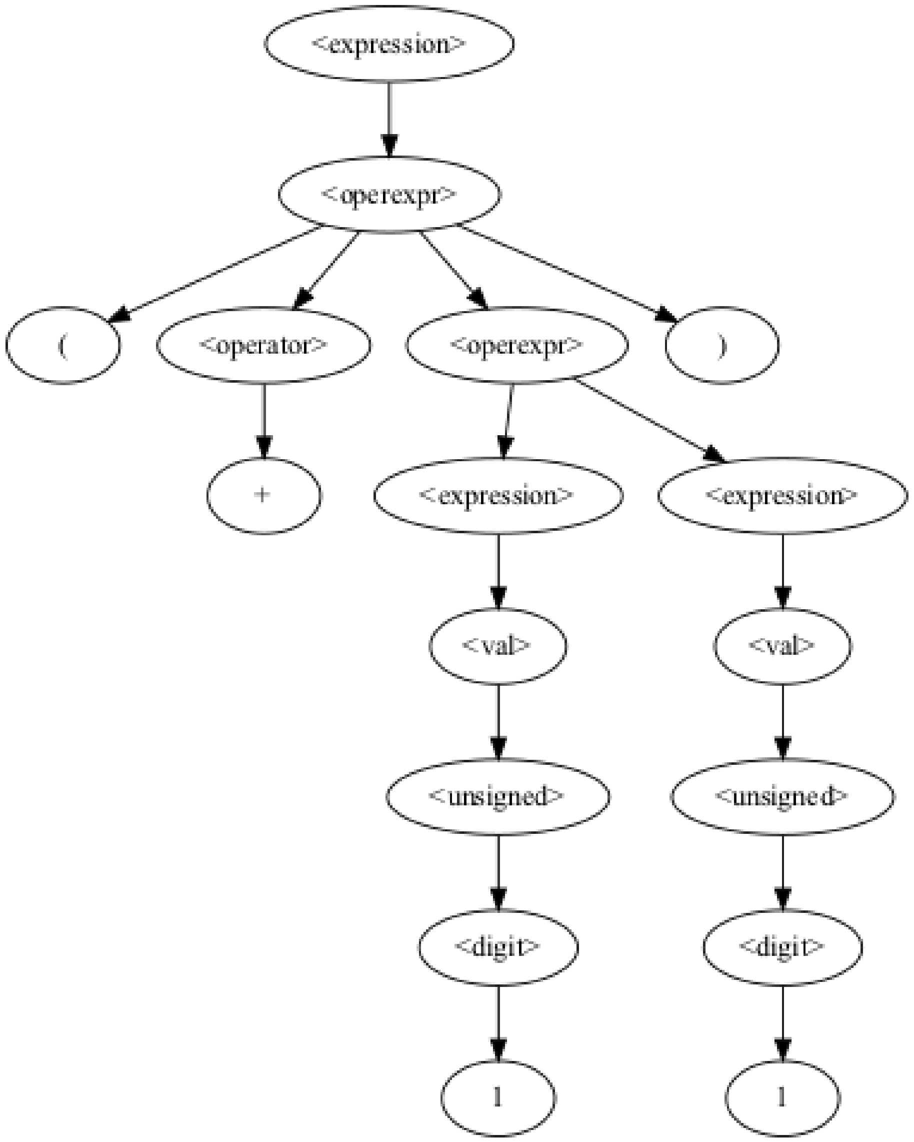

Figure 41.3: Parse tree for prefix (+ 1 1).

- the Lua-based Expression Language 1 and Imperative Core interpreters I developed for the Fall 2016 CSci 450 course
- chapters 1, 2, and 4 of Samuel Kamin's textbook *Programming Languages: An Interpreter-Based Approach* [6] and my work to implement three (Core, Lisp, and Scheme) of Kamin's interpeters in Lua in 2013
- sections 8.3 and 9.6 of the classic Richard Bird and Philip Wadler's textbook *Introduction to Functional Programming* [2]
- sections 14.2, 16.1, 17.5, and 18.3 of Simon Thompson's textbook [10]
- chapters 1-4 and 8 of Peter Sestoff's textbook *Programming Language Concepts* [9]
- chapters 21 (Recursive Descent Parser) and 22 (Parser Combinator) of Martin Fowler and Parsons's book *Domain-Specific Languages* [5].
- section 3.2 (Predictive Parsing) of Andrew W. Appel's textbook *Modern Compiler Implementation in ML* [1].
- chapters 6 (Purely Functional State) and 9 (Parser Combinators) from Paul Chiusano and Runar Bjarnason's *Functional Programming in Scala* [3].
- sections 1.2, 3.3, and 5.1 of Peter Linz's textbook *Formal Languages and Automata* [7]
- the Wikipedia articles on Formal Grammar [11], Linear Grammar {[15]}, Regular Grammar [16], Context-Free Grammar [12], Backus-Naur Form [13], Extended Backus-Naur Form [14], Parsing [21], Parse Tree [20], Recursive Descent Parser [23], LL Parser [22], Lexical Analysis [26], Finite-state Machine [17], Deterministic Finite Automaton [18], Pushdown Automaton [19], Abstract Syntax [24], Abstract Syntax Tree [25], Stack Machine [27], Reverse Polish Notation [28], Association List [29], and Associative Array [30].

For the 2017 textbook, I organized this work into three chapters:

10. Expression Language Syntax and Semantics
11. Expression Language Parsing
12. Expression Language Compilation (a partial chapter)

In Summer 2018, I divided the previous Expression Language Syntax and Semantics chapter into three new chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming.*

- Previous section 10.2 became new Chapter 41, Calculator Concrete Syntax.
- Previous sections 10.3-5 and 10.7-8 became new Chapter 42, Calculator Abstract Syntax and Evaluation.

- Previous sections 10.6 and 10.9 became new Chapter 43, Calculator Modular Structure, and were expanded.

In Fall 2018, I divided the 2017 Expression Language Parsing chapter into two new chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming.*

- Previous sections 11.1-11.4 became new Chapter 44, Calculator Parsing.
- Previous sections 11.6-11.7 became new Chapter 45, Parsing Combinators.
- Previous section 11.5 was merged into new Chapter 43, Calculator Modular Structure.

In Fall 2018, I also renumbered previous chapter 12 to become new Chapter 46.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on the ELIFP 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.

41.10 Terms and Concepts

Syntax, concrete syntax, formal grammar (variable and terminal symbols, alphabet, start or goal symbol), production rule, sentence, sentential form, language, context-free grammar, Backus-Naur Form (BNF), derivation, leftmost derivation, derivation tree, right-lean and right-linear grammar, regular grammar, deterministic finite automaton (DFA), pushdown automaton (PDA), infix and prefix syntaxes, lexical level, parsing, parser, parse tree, infix and prefix syntax.

41.11 References

[1] Andrew W. Appel. 1998. *Modern compiler implementation in ML.* Cambridge University Press, Cambridge, UK.

[2] Richard Bird and Philip Wadler. 1988. *Introduction to functional programming* (First ed.). Prentice Hall, Englewood Cliffs, New Jersey, USA.

[3] Paul Chiusano and Runar Bjarnason. 2015. *Functional programming in Scala* (First ed.). Manning, Shelter Island, New York, USA.

[4] H. Conrad Cunningham. 2019. *Notes on Scala for Java programmers.* University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/csci555/notes/ScalaForJava/ScalaForJava.html

[5] Martin Fowler and Rebecca Parsons. 2010. *Domain specific languages.* Addison-Wesley, Boston, Massachusetts, USA.

[6] Samuel N. Kamin. 1990. *Programming languages: An interpreter-based approach.* Addison-Wesley, Boston, Massachusetts, USA.

[7] Peter Linz. 2017. *Formal languages and automata* (Sixth ed.). Jones & Bartlett, Burlington, Massachusetts, USA.

[8] Michel Schinz and Phillipp Haller. 2016. A Scala tutorial for Java. Retrieved from https://docs.scala-lang.org/tutorials/scala-for-java-programmers.html

[9] Peter Sestoft. 2012. *Programming language concepts* (First ed.). Springer, London, UK.

[10] Simon Thompson. 2011. *Haskell: The craft of programming* (Third ed.). Addison-Wesley, Boston, Massachusetts, USA.

[11] Wikpedia: The Free Encyclopedia. 2022. Formal grammar. Retrieved from https://en.wikipedia.org/wiki/Formal_grammar

[12] Wikpedia: The Free Encyclopedia. 2022. Context-free grammar. Retrieved from https://en.wikipedia.org/wiki/Context-free_grammar

[13] Wikpedia: The Free Encyclopedia. 2022. Backus-naur form. Retrieved from https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form

[14] Wikpedia: The Free Encyclopedia. 2022. Extended backus-naur form. Retrieved from https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form

[15] Wikpedia: The Free Encyclopedia. 2022. Linear grammar. Retrieved from https://en.wikipedia.org/wiki/Linear_grammar

[16] Wikpedia: The Free Encyclopedia. 2022. RegularGrammar. Retrieved from https://en.wikipedia.org/wiki/Regular_grammar

[17] Wikpedia: The Free Encyclopedia. 2022. Finite-state machine. Retrieved from https://en.wikipedia.org/wiki/Finite-state_machine

[18] Wikpedia: The Free Encyclopedia. 2022. Deterministic finite automaton. Retrieved from https://en.wikipedia.org/wiki/Deterministic_finite_automaton

[19] Wikpedia: The Free Encyclopedia. 2022. Pushdown automaton. Retrieved from https://en.wikipedia.org/wiki/Pushdown_automaton

[20] Wikpedia: The Free Encyclopedia. 2022. Parse tree. Retrieved from https://en.wikipedia.org/wiki/Parse_tree

[21] Wikpedia: The Free Encyclopedia. 2022. Parsing. Retrieved from https://en.wikipedia.org/wiki/Parsing

[22] Wikpedia: The Free Encyclopedia. 2022. LL parser. Retrieved from https://en.wikipedia.org/wiki/LL_parser

[23] Wikpedia: The Free Encyclopedia. 2022. Recursive descent parser. Retrieved from https://en.wikipedia.org/wiki/Recursive_descent_parser

[24] Wikpedia: The Free Encyclopedia. 2022. Abstract syntax. Retrieved from https://en.wikipedia.org/wiki/Abstract_syntax

[25] Wikpedia: The Free Encyclopedia. 2022. Abstract syntax tree. Retrieved from https://en.wikipedia.org/wiki/Abstract_syntax_tree

[26] Wikpedia: The Free Encyclopedia. 2022. Lexical analysis. Retrieved from https://en.wikipedia.org/wiki/Lexical_analysis

[27] Wikpedia: The Free Encyclopedia. 2022. Stack machine. Retrieved from https://en.wikipedia.org/wiki/Stack_machine

[28] Wikpedia: The Free Encyclopedia. 2022. Reverse polish notation. Retrieved from https://en.wikipedia.org/wiki/Reverse_Polish_notation

[29] Wikpedia: The Free Encyclopedia. 2022. Association list. Retrieved from https://en.wikipedia.org/wiki/Association_list

[30] Wikpedia: The Free Encyclopedia. 2022. Associative array. Retrieved from https://en.wikipedia.org/wiki/Associative_array