A Monadic Combinator Compiler Compiler

Carlos Camarão and Lucília Figueiredo

February 2, 2001

Abstract

This article describes a novel approach to compiler generation, based on monadic combinators. A prototype of a compiler generator, called *Mímico*, is described, that handles infinite look-ahead and left recursive context free grammars, and dyadic infix operator precedence and associativity. Novel ideas and the general principle from which this work has been based are presented, as well as limitations of Mímico and a comparison with related work.

1 Introduction

This article describes a prototype of a monadic combinator compiler compiler, called *Mímico*. Mímico can work with left recursive context free grammars and allows for an easy specification of precedence and associativity of dyadic infix operators.

Some familiarity with functional programming and the language Haskell[Bir98, Hud00] is assumed, as well as with monadic programming[Wad92a, Wad90, Hut92, HM96].

Section 2 describes the input, output, overall behaviour and structure, as well as the general principle behind Mímico, by means of simple examples. Subsequent sections discuss the treatment of left recursion, and operator precedence and associativity, respectively. Section 5 synthesizes limitations of Mímico and of compiler generation based on the use of monadic combinators in general. Section 6 provides a comparison with related work. Section 7 concludes.

2 Overview

Mímico is a Haskell program prototype comprised of four modules, together with module *Parser* from the combinator library *Parsec*[Lei00].

- *Grammar* contains type defintions and functions for treating context free grammars (*first*, *follow* and other related functions).
- *Prod* is responsible for reading the input file and generating the context free grammar description, with associated semantic rules, as defined in module *Grammar*.
- *Info* checks input validity and collects information needed for compiler generation.
- *Mimico* contains the main routines for compiler generation, taking the result of the grammar description, annotated with data generated from module *Info*.

Module *Parser* defines usual parsing combinators (*many*, *manyTill*, *choice* and others), parsers for lexical analysis (i.e. character parsers like *letter*, *digit*, *lower*, *upper*, *space*, *spaces*, *char*, *anyChar* and *string*), and functions for the support of better error message reporting (mainly <?>; see [Lei00]). The primary reason for using Parsec, instead of another combinator library, was support for error handling (see [Lei00]). However, programs generated by Mímico use another set of combinators, taken from [Bir98], because Parsec's parser type does not provide support for nondeterministic parsing (see below).

As is well-known[Wad92b, Hut92, Wad90, HM96], a monadic parser can be viewed as a function that takes a string as input and yields a list of pairs: the first component represents the result of parsing the consumed part of the input string, and the second component represents the unconsumed part of the input after parsing.

Taken into account this definition, our approach to compiler generation is based on the following basic informal rule:

> Alternatives for a given nonterminal are parsed according to their textual order in the input file, meaning that those that should occur at higher levels of the generated parse tree should be written before those at lower levels.

We can refer to this rule as "first alternatives higher in the parse tree" (or, simply, *first alts higher*). Consider, as an example, the following simple input grammar:

```
A = a       { "yes" }
  | a A     { "yes" }
```

Mímico does not require the user to explicitly distinguish terminals from nonterminals, neither to specify which is the initial nonterminal symbol. Nonterminals are those symbols which appear in the left hand sides (lhs) of the input grammar, and the initial nonterminal is the nonterminal at the lhs of the first production.

Productions finish according to a layout rule: a next production has the same indentation as its previous one. The first `=` occurs after the non-terminal, and symbol `|` indicates the start of an alternative and should occur in the same column as the first `=`.

This simple grammar should *not* be given as input to Mímico, on intent of generating a program that outputs `"yes"` when given as input a sequence of `a`'s, and fails (issuing an error message) otherwise. The program generated by Mímico for this grammar will successfully parse *all* the input only if this input consists of a unique symbol `a`.

The input grammar for $\mathtt{a}^*$ can be written as follows:

```
A = a A       { "yes" }
  | a         { "yes" }
```

The parser for the first alternative tries to parse more than one `a`. If it fails, the second alternative is tried. Given this input, Mímico generates the following:

```
import Parser
compile = apply a1

a1 = do symb "a"
        a_1 <- a1
        return "yes"
     <|> a2

a2 = do symb "a"
        return "yes"
```

Monadic parser generation involves the simple general idea of syntax-directed recursive descent parsing: a parser is written for each production, which is a combination of parsers for its constituent symbols. A monadic *compiler* is formed by simply following the corresponding monadic parser by a (monadic) return action. This return has as parameter the Haskell code that constitutes the semantic rule (possibly with some adaptations, as we will see in the sequel).

In the above example, compilers `a1` and `a2` correspond to each alternative of the production, whose lhs is nonterminal `A`. The names of each compiler are generated from the name of the nonterminal at the lhs, followed by the alternative number. If, as in this example, the nonterminal starts with an upper case letter, Mímico tests whether there exists a nonterminal with the same name, but with the first letter changed to

lower case (since function names may not start with an upper case letter in Haskell); if this new name is not already used as the name of another nonterminal, it is chosen as the compiler name; otherwise, character `'_'` is introduced before the name of the nonterminal to form the compiler name.

The parser for a terminal symbol `s` is one of the (usual) lexical analysers `string s` or `symb s`. Which one depends on the following. We call *lexeme* a grammar symbol that may be followed by *white spaces* (i.e. either blanks, newlines or tabs), and use the following conventions:

- if the name of a nonterminal symbol starts with a lowercase letter, then it is a lexeme. For example, in the right hand side (rhs) `e + e`, nonterminal symbol `e` is a lexeme, and in the rhs `a A`, nonterminal symbol `A` is not.
- if a terminal symbol is followed by another terminal or by a nonterminal which is a lexeme, then it is a lexeme. For example, in the rhs `e + e`, terminal symbol `+` is a lexeme, and in the rhs `a A`, terminal symbol `a` is not.

Combinator `<|>` implements nondeterministic choice: `p <|> q` applies both `p` and `q` and appends their results. Lazy evaluation is clearly important here, for the sake of efficiency. Nondeterministic parsing is not needed in most cases, but it allows the use of some non-predictive (not LL(k)) grammars, as in the example at the end of this section.

`apply` is the simple monadic deconstructor (or "`run`") function; that is, the result of `apply (Parser p) s` is the first component of the resulting list given by `p s`.

The semantic rules, specified after each rhs of an alternative, can be used, of course, for more interesting purposes, other than merely specifying a (kind of) language recognizer. In the next example, we count the number of `a`'s in the input sequence:

```
A = a A { 1 + A }
  | a   { 1 }
```

```
import Parser
compile = apply a1

a1 = do string "a"
        a_1 <- a1
        return ( 1 + a_1 )
     <|> a2

a2 = do { string "a"; return 1 }
```

The variable that receives the result of the compilation and the compiler names corresponding to nonterminal `A` are, respectively, `a1` and `a_1`.

Mímico adopts the following simple conventions for the specification and interpretation of semantic rules:

- the absence of a semantic rule is interpreted in the same way as one textually identical to the rhs. Briefly, *no semantic rule* means (*semantic rule = rhs*).

- if a nonterminal `v` occurs more than once in a rhs, each occurrence is distinguished in the semantic rule by specifying subscripts `v_1`, `v_2` etc., where the subscripts (which must be numbers) refer to the order of occurrence of `v` in the rhs.

  For example, production `e = e + e { e_1 + e_2 }` specifies `e_1` as the result of the compilation of the first occurrence of nonterminal `e` in the rhs, and `e_2` as the result of the compilation of the second occurrence of `e`.

- if a nonterminal `v` occurs more than once in a rhs and a subscript is not used in the semantic rule to distinguish distinct occurrences of this nonterminal, as explained above, than these occurrences in the semantic rule are interpreted as numbered from 1 upwards, according to its textual order of occurrence in the semantic rule.

  For example, `e = e + e { e + e }` is equivalent to `e = e + e { e_1 + e_2 }`. Production `e = e + e { e + e + e }` is incorrect (an error is detected when this production is included in an input grammar). Production `e = e + e { e_1 + e + e_2 + e}` is equivalent to `e = e + e { e_1 + e_1 + e_2 + e_2}`.

We present next a final introductory example, that illustrates the use and need for nondeterministic parsing, that allows infinite look-ahead (non-predictive) grammars as input. The example is of a simple grammar for recognizing palindromes:

```
P = Char P Char { Char == Char }
  | Char        { True }
  | epsilon     { True }
```

`Char` and `epsilon` are reserved nonterminals, representing any symbol and an empty rhs, respectively. The program generated by Mímico is shown in Figure 1.

The use of the nondeterministic choice operator `<|>` allows input strings such as `aa` to be parsed successfully. Parsing this input involves a "backtrack", on the recursive call to `p1`. The first call to `p1` consumes the first input symbol `a` before calling `p1` again. This recursive call suceeds, returning the list of sucessful parses `[("a",""),("","a")]`. Of these, the first does not originate a successful parse for the first call to `p1`; but the second (after a "backtrack"), in which no input is consumed on the recursive call, does originate a successful parse of the input.

The next sections describe the ideas that enabled us to develop Mímico's handling of left recursion (Section 3) and operator precedence and associativity (Section 4).

```
import Parser
compile = apply p1

p1 = do char_1 <- sat (const True)
        p_1 <- p1
        char_2 <- sat (const True)
        return ( char_1 == char_2 )
     <|> p2

p2 = do char_1 <- sat (const True)
        return True
     <|> return True
```

Figure 1: Nondeterministic parsing handling an infinite look-ahead input grammar

3 Left Recursion

Since monadic parsing is based on a recursive descent technique, one might expect that left-recursive productions are not allowed as input (because they would cause monadic parsers to go into an infinite loop). Furthermore, one could also think that it would be natural to take into account the usual argument that left-recursive grammars can be rewritten to an equivalent non-left-recursive grammar. However, this is not a strong argument in the case of a compiler generator, since the non-left-recursive grammar may not be so simple to specify as its left-recursive counterpart, specially with respect to the semantic rule, which may also turn out to specify a less efficient algorithm (see below).

Consider as a first example the following left recursive grammar, describing the language $\{\mathtt{ab}^*\}$:

```
X = X b { X + 1 }
  | a   { 0 }
```

The semantic rules specify the output to be the number of `b`'s in a given input. Given this input grammar, Mímico generates the program shown in Figure 2.

In this example, left recusion is eliminated by using essentially the standard left-recursion elimination algorithm for context-free grammars[AVAU86]. We'll see in Section

```
import Parser
compile = apply x1

x1 = do _b_1 <- _b1
        x'_1 <- x'1
        return ( let f z x = z + 1 in foldl1 f ( _b_1 :  x'_1 ) )

_b1 = do { string "a"; return 0 }

x'1 = do _undef_1 <- _undef1
         x'_1 <- x'1
         return ( _undef_1 :  x'_1 )
      `orelse` return []

_undef1 = do { string "b"; return undefined }
```

Figure 2: Example of program generated from a left recursive grammar

4 that left recursion is not always handled with this approach, with the aim of simplifying the specification of the input grammar, operator associativity in particular.

Because of semantic rules, left recursive productions must have the simpler form:

$$\begin{array}{ll} \mathtt{A} = \mathtt{A}\ \mathtt{R}_i & \{\ \mathtt{sr}_i\ \} \\ \quad \mid \mathtt{b}_j & \{\ \mathtt{sr}'_j\ \} \end{array}$$

where $\mathtt{A}\ \mathtt{R}_i\{\mathtt{sr}_i\}$ is an abbreviation for alternatives $\mathtt{A}\ \mathtt{R}_1\{\mathtt{sr}_1\} \mid \ldots \mid \mathtt{A}\ \mathtt{R}_n\{\mathtt{sr}_n\}$ and analogously for $\mathtt{b}_j\{\mathtt{sr}'_j\}$, each $\mathtt{R}_i$ is a sequence of symbols containing at least one symbol and at most one nonterminal symbol, and each $\mathtt{b}_j$ is any sequence of symbols not beginning with $\mathtt{A}$. Formally $\mathtt{R}_i = u\mathtt{S}_i v$ or $\mathtt{R}_i = w$, where $u, v \in \Sigma^*$, $w \in \Sigma^+$, $\mathtt{S}_i \in V$, Σ is the set of terminal symbols and V the set of nonterminal symbols.

Semantic rules not withstanding, it is trivial to transform any context free grammar into this form; this restriction is relevant only with respect to the requirement that semantic rule $\mathtt{sr}_i$ be written with respect to $\mathtt{A}$ and $\mathtt{S}_i$ (for $i = 1, \ldots, m$).

This left recursive form is converted to the following (for $i = 1, \ldots, n$, $j = 1, \ldots, m$), where abbreviations are used similarly as before:

```
A  = B A'          { code_i }

B  = b_j           { sr'_j }

A' = S'_i A'       { S'_i:A'}
   | epsilon       { [] }

S'_i = R_i         { S'_i }
```

where `A'` and `B` and each S'_i are fresh nonterminal symbols, and, for each i, $\texttt{code}_i$ is given by (where `sr[B/A]` denotes the string obtained from `sr` by subsitituting `B` for `A`):

```
foldl1 f (B: A')
       where f z x = sr_i [z/A, x/S_i]
```

Also note the following: for each j, when $\{b_j\}$ consists of a single nonterminal, production $\texttt{B = b}_j$ is not needed; for each i, when $\texttt{R}_i$ is a single nonterminal, production $\texttt{S}'_i = \texttt{R}_i$ is not neeeded and, furthermore, when $\texttt{R}_i = w$ (that is, there exists no nonterminal after the left recursive nonterminal `A`), $\texttt{S}'_i$ is (set to) `undefined`.

This translation, with the use of `foldl1`, can be seen as a formal specification of the semantics of semantic rules for left-recursive grammars. Informally, the input is parsed and the results of compilations are collected into a list, say l. The basic point to note is that the semantic rule of the i-th alternative, in the original left-recursive grammar, considers the list of results l as given by $\texttt{[A}_1\texttt{, ... , A}_k\texttt{, S}_i\texttt{]}$, where $\texttt{A}_1$, ... , $\texttt{A}_k$ are the results of compilations of the left-recursive nonterminal `A`. Therefore, folding over this list to obtain the desired compilation result is given by `foldl1 f l`, where `f` is defined as above. This list is formed, in the non-left-recursive grammar, by successively applying the right-recursive productions of $\texttt{A}'$ after the base case given by `B`. The base case corresponds to applying one of the productions $\texttt{b}_j$, for some $j \in \{1, \ldots, m\}$.

The next example illustrates that left-recursion may indeed simplify the specification of the semantic rules. The following grammar transforms binary into decimal notation:

```
L = L B        { B + 2*L }
  | B

B = 0          { 0 }
  | 1          { 1 }
```

```
import Parser
compile = apply l1

l1 = do b_1 <- b1
        l'_1 <- l'1
        return ( let f z x = x + 2 * z in foldl1 f ( b_1 : l'_1 ) )

l'1 = do b_1 <- b1
         l'_1 <- l'1
         return ( b_1 :  l'_1 )
      <|> return []

b1 = do { string "0"; return 0 }
     <|> b2

b2 = do { string "1"; return 1 }
```

Figure 3: Program generated for converting binary to decimal notation

Note that again no semantic rule is needed for the second alternative of nonterminal `L` (*no semantic rule* means *semantic rule* = *rhs*). Mímico generates the program shown in Figure 3 for this grammar. Left recursion not only simplifies the semantic rule in this case; it also leads to the generation of more efficient code. Compare with the right-recursive alternative:

```
L = B L       { ( B * 2^(snd L) + fst L, snd L + 1) }
  | B         { (B, 1) }

B = 0         { 0 }
  | 1         { 1 }
```

4 Precedence and Associativity

Consider now the following archetypical left-recursive grammar for arithmetic expressions (we use only the symbols `+`, `-` and `*` for brevity):

```
e = e + e
  | e - e
  | e * e
  | n
  | (e)   { e }
```

`n` is a reserved meta-variable, denoting an integer numeral lexeme. Note the absence of semantic rules (*no semantic rule* means *semantic rule = rhs*).

Given this concise input, Mímico generates the expression evaluator shown in Figure 5 (with program header removed, for brevity). We will look at this program in a moment; before doing so, however, let us examine a simpler expression evaluator program, generated for the following similar input:

```
e =.  e + e
  |.  e - e
  |.  e * e
  | n
  | (e)     { e }
```

The dots inserted in this grammar indicate right-associativity of dyadic operators. The grammar specifies also, according to the *first alts higher* rule, that `+` has the lowest precedence, then `-` and finally `*`. The program generated is shown in Figure 4.

Each of the little compilers `e1` to `5` correspond to an alternative of the input grammar. Compilers `e1`, `e2` and `e3` are parameterised, to control operator precedence. The parameter indicates a list of terminals that should not be parsed successfully, because they should be parsed using a parser of lower precedence level. Right-associativity follows directly, in this scheme. Left-associativity, the default, is based on the same parametrization scheme but uses left-factoring (see Figure 5).

It is possible to "override" the *first alts higher* rule and specify that infix operators should have the same precedence, and left or right associativity. For example:

```
e = e + e
  |= e - e
  | e * e
  | n
  | (e)   { e }
```

```
e = e + e
  =| e - e
  | e * e
  | n
  | (e)   { e }
```

```
import Parser
import Char
import List
compile = apply (e1[])

e1 s = do e_1 <- token $ e2(["+"] `union` s)
          if "+" `elem` s then zerop else symb "+"
          e_2 <- token $ e1 s
          return ( e_1 + e_2 )
       <|> e2 s

e2 s = do e_1 <- token $ e3(["-"] `union` s)
          if "-" `elem` s then zerop else symb "-"
          e_2 <- token $ e1 s
          return ( e_1 - e_2 )
       <|> e3 s

e3 s = do e_1 <- token $ e4
          if "*" `elem` s then zerop else string "*"
          e_2 <- token $ e1 s
          return ( e_1 * e_2 )
       <|> e4

e4 = do n_1 <- token (some (sat isDigit))
        return ( read n_1 )
     <|> e5

e5 = do string "("
        e_1 <- token $ e1 []
        string ")"
        return e_1
```

Figure 4: *First alts higher* and right associativity

```
e1 s = do e_1 <- token $ e2(["+"] `union` s)
          op <- if "+" `elem` s then zerop else do symb "+"; return (+)
          rest1 s e_1 op
       <|> e2 s

rest1 s e_1 op0 = do e_2 <- token $ e2(["+"] `union` s)
                     op <- if "+" `elem` s then zerop
                           else do symb "+"; return (+)
                     let opnd = op0 e_1 e_2
                     rest1 s opnd op
                  <|> do e_2 <- token $ e1 s
                         return ( op0 e_1 e_2 )

e2 s = do e_1 <- token $ e3(["-"] `union` s)
          op <- if "-" `elem` s then zerop else do symb "-"; return (-)
          rest2 s e_1 op
       <|> e3 s

rest2 s e_1 op0 = do e_2 <- token $ e3(["-"] `union` s)
                     op <- if "-" `elem` s then zerop
                           else do symb "-"; return (-)
                     let opnd = op0 e_1 e_2
                     rest2 s opnd op
                  <|> do e_2 <- token $ e1 s
                         return ( op0 e_1 e_2 )

e3 s = do e_1 <- token $ e4
          if "*" `elem` s then zerop else string "*"
          e_2 <- token $ e1 s
          return ( e_1 * e_2 )
       <|> e4

e4 = do { n_1 <- token (some (sat isDigit)); return ( read n_1 ) }
     <|> e5

e5 = do { string "("; e_1 <- token $ e1 []; string ")"; return e_1 }
```

Figure 5: Left factoring for left associativity

Symbols `|=` and `=|` mean left and right associativity, respectively, with respect to the operator in the previous alternative.

The programs generated in these cases differ only slightly from the corresponding ones above (for left and right associativity, respectively), as shown in Figure 6. Compilers for alternatives with same precedence are coalesced into a single one, and the arguments passed to compilers for the next alternative are the union of the current list with the list of terminals with same precedence (cf. Figure 4).

Similar programs are also generated if a nonterminal is used in the place of the infix operators, as in the following grammar:

```
e = e op e        { op e e }
  | n
  | (e)           { e }

op = +
   | -
   | *
```

5 Limitations and Further Work

Mímico is still a prototype. What is mostly needed is better error handling and reporting. In the current implementation, if the input is syntactically wrong, the program simply halts, issuing a simple error message. There is not yet any mechanism that enables the user to specify specific error messages for certain kinds of input.

Another aspect is related to providing the possibility for the user to override the *first alts higher* rule, for example, for allowing ambiguous nested *dangling elses* to be associated to the outermost conditional expression (or command).

Further work includes analysis of the runtime efficiency of programs generated by Mímico, with a comparison with other compiler generators, like e.g. Yacc[LMB92] and Happy[GM00].

6 Related Work

A lot of work has been done on compiler generators[LMB92, Lee89, MM00, GM00]. We have not yet done any performance comparisons with well-known generators like Yacc[LMB92] and Happy[GM00], because Mímico is still in its infancy. Our main motivation up to now has been to show that it is possible and worthwhile to generate simple

```
import Parser
import Char
import List
compile = apply (e1[])

e1 s = do e_1 <- token $ e3(["+","-"] `union` s)
          op <- if "+" `elem` s && "-" `elem` s then zerop
                else do symb "+"; return (+) <|> do symb "-"; return (-)
          rest s e_1 op
       <|> e3 s

rest s e_1 op0 = do e_2 <- token $ e1 s
                    return ( op0 e_1 e_2 )

e3 s = do e_1 <- token $ e4
          if "*" `elem` s then zerop else string "*"
          e_2 <- token $ e1 s
          return ( e_1 * e_2 )
       <|> e4

e4 = do n_1 <- token (some (sat isDigit))
        return ( read n_1 )
     <|> e5

e5 = do string "("
        e_1 <- token $ e1 []
        string ")"
        return e_1
```

Figure 6: Same precedence and right associativity

and readable monadic compilers automatically, from an input grammar, written in a clear way and without the need to separately specifying lexical analysers. We expect the generated compilers to be reasonably efficient, in comparison to (say) Happy. This will not be surprising, since monadic parsing and lexers are known to be fast, with run-time efficiency comparable to parsers generated by Happy, which are based, as parsers generated by Yacc and many others, on the LALR(1) parsing technique.

One particular point has encouraged us to do this work with enthusiasm. Mímico input grammars and output programs are remarkably simple and readable, in comparison to those produced by Yacc and Happy, and this is not because our examples are small. We are confident and will emphasize this aspect as we see Mímico getting older and smarter.

Another point that is well worthwhile pointing out is the fact that Mímico allows as input grammars that are non-predictive (i.e. not LL(k), for any k). For example, the simple grammar for palindromes presented at the introduction would not be accepted by Yacc or Happy.

A simple and interesting question — for which we do not yet know the answer — is which condition should input grammars satisfy such that its parsing (in general, and, also, using our parsing scheme generation in particular) requires the use of a non-deterministic parser combinator? Note that deterministic parser combinators can be used (by Mímico) to parse even some non-predictive grammars, such as, for example, $\{a^n 0 b^{2n}\} \cup \{a^n 1 b^n\}$.

7 Conclusion and Further Work

We have described a novel approach to compiler generation, based on monadic combinators, and have described a prototype implementation, called Mímico. As far as we know, this is the first compiler compiler based on monadic parsing. We have presented the general principle (*first alts higher* rule) and the ideas on which this work has been based (the scheme for supporting left recursive input grammars, parametrization of compilers as a way of handling operator precedence and associativity, and the use of nondeterministic parsing to allow non-predictive grammars as input). Noticeable contributions of Mímico, when comparing with other related work, is the clarity and readability of Mímico's input grammars and output programs, as well as the fact that Mímico accepts as input and generates programs that are able to parse even non-predictive grammars.

References

[AVAU86] Ravi Sethi Alfred V. Aho and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice-Hall, 1998. 2nd ed.

[GM00] Andy Gill and Simon Marlow. Happy: The Parser Generator for Haskell. http://haskell.org/happy/, 2000.

[HM96] Graham Hutton and Erik Meijer. Monadic parser combinators. Tech. rep. NOTTCS-TR-96-4, 1996.

[Hud00] Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000.

[Hut92] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2:232–343, 1992. http://www.cs.nott.ac.uk/Department/Staff/gmh/parsing.ps.

[Lee89] P. Lee. *Realistic Compiler Generation*. MIT Press, 1989.

[Lei00] Daan Leijebn. Parsec, a fast combinator library. http://www.cs.ruu.nl/ daan/parsec.html, 2000.

[LMB92] John R. Levine, Tony Mason, and Doug Brown. *Lex & Yacc*. O'Reilly & Associates, 1992.

[MM00] Alexandre Macedo and Hermano Moura. Investigating Compiler Generation Systems. In *Proc. SBLP'2000 (IV Brazilian Symposium on Programming Languages*, pages 259–266, 2000.

[Wad90] Philip Wadler. Comprehending Monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1990.

[Wad92a] Philip Wadler. Monads for Functional Programming. *Computer and Systems Sciences*, 118, 1992. http://www.cs.bell-labs.com/who/wadler/topics/monads.html.

[Wad92b] Philip Wadler. The essence of functional programming. In *Conference Record of the 19th ACM Symposium on Principles of Programming Languages*, pages 1–14, 1992.