Mímico: A Monadic Combinator Parser Generator

Carlos Camarão

Universidade Federal
de Minas Gerais
31270-010 - Belo Horizonte
MG - Brasil
camarao@dcc.ufmg.br

Lucília Figueiredo

Universidade Federal
de Ouro Preto
35100-000 - Ouro Preto
MG - Brasil
lucilia@dcc.ufmg.br

Hermann Rodrigues

Universidade Federal
de Minas Gerais
31270-010 - Belo Horizonte
MG - Brasil
hermann@dcc.ufmg.br

Abstract

This article describes a compiler generator, called *Mímico*, that outputs code based on the use of monadic combinators. Mímico can parse infinite lookahead and left-recursive context free grammars and defines a scheme for handling the precedence and associativity of binary infix operators, and monadic code in semantic rules. Mímico provides an easy way of specifying the syntax and semantics of languages, and generates readable output in the form of Haskell programs. The article presents Mímico's general principles, its formal syntax and semantics, its limitations and illustrative examples of its behaviour.

Keywords: *Compiler generation, monadic parsing*

1 Introduction

This article describes a compiler generator, called *Mímico*, that outputs code based on monadic combinators. The paper assumes familiarity with Haskell [2, 15] and monadic programming [18, 17, 6, 7]. A preliminary and shorter version of this paper, which does not include, in particular, a formalization of the semantics of Mimico's input grammars (Section 2) and a description of the support for the generation of monadic code (Section 4), has been published in the SBLP'2001 Conference Proceedings [3].

The input to Mímico is a grammar specification where productions have the form $\mathtt{A}_i = r_i$ {| e_i |}, for $i = 1, \ldots, n$, where $\mathtt{A}_i$ is a nonterminal symbol, called the production's left-hand side (lhs), r_i is a sequence of terminal and nonterminal symbols, called the production's right-hand side (rhs), and e_i is a Haskell expression, called the production's semantic rule.

The output generated by Mímico is a recursive descent parser, based on monadic combinators, that parses strings generated by the input grammar. A parser is generated for each production, as a combination of parsers for nonterminal symbols occurring in the rhs of the production. In fact, Mímico generates monadic compilers, by simply following monadic parsers by a (monadic) return action. This return action has as parameter the Haskell code that constitutes the semantic rule (possibly with some adaptations, as we will see in the sequel).

The clarity of the code written in Haskell, the possibility to work with higher-order functions and, finally, lazy evaluation, are basic fundamental tools used by Mímico in order to be able to accept grammars as input and generate as output programs that "mimic" the input specification. Due to the readability of its input grammars, which involve the use of Haskell to specify the semantic rules, we hope that Mímico can become a useful tool for teaching the formal specification of the syntax and semantics of languages.

Mímico is also based on the following informal rule: *alternatives for a nonterminal are parsed in their textual order*, called the *Pastor rule* ("parse as specified by the textual order"). It means that an automatically generated parser tries to parse a subsequent alternative only if the previous ones have failed to produce a successful parse of the input. Of course, a successful parse for a given nonterminal may involve parsing not all but only part of the input.

The Pastor rule removes all sources of ambiguities in grammar specifications, by considering grammar productions for each nonterminal as a list, instead of as a set. For grammars which are ambiguous, considering productions as a set, the parsing tree implicitly produced by the parser generated by Mímico is determined by this textual order. This may also have an impact on the efficiency of the generated parser. Examples illustrating these aspects are given below.

Informally, the behaviour of Mímico can be described as follows: for a *list* of productions $\mathbb{P}$ = [$\mathtt{A}_i = r_i$ {| e_i |}], for $i = 1, \ldots, n$, Mímico generates compilers which follow the Pastor rule for all productions in $\mathbb{P}$

and a compiler for an initial symbol S corresponding to the following additional production, automatically introduced by Mímico: S = A_1 eoi {| A_1 |} , where eoi denotes a parser that succeeds if and only if the end of the input is reached. (and A_1 is the original initial symbol of the grammar). An example at the end of this section illustrates the need of testing the end of the input in the generated compiler in this additional production.

Consider, as a first example, the following simple input grammar for Mímico:

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

In this paper we consider that symbols that appear in left hand sides (lhs) of input grammar productions are nonterminals, and the initial nonterminal is the one at the lhs of the first production. This form of distinguishing terminals from nonterminals is useful for small examples and test cases — and can be explicitly chosen by a command-line flag, when starting Mímico —, but is not adequate from a software engineering point of view. This occurs because the inclusion of a new rule can change the effect of existing rules, if the nonterminal at the lhs of the new rule appears as a terminal in an existing rule.

Also, by default alternatives in a production finish according to a layout rule: a next alternative has the same indentation as its previous one, or is indented more (to the right). The first = occurs after the lhs nonterminal, and symbol | indicates the start of an alternative. It should occur in the same column as the first =, or after this column. The layout rule can be switched off by a flag (-fNoLayout) when starting Mímico. In this case, a sequence of alternative productions must end with symbol ';' (which is, with -fNoLayout, a reserved symbol). A reserved symbol or a sequence of reserved symbols must be preceded by a backslash ('\'), to be written in the rhs of productions or in the semantic rule, and a backslash must itself be written as '\\'.

The parser for the first alternative in the grammar above will try to parse more than one a; if it fails, the second alternative will be tried. Given this input, Mímico generates the program shown in Figure 1.

```
import Parser

compile = apply a

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

a1 = do symb "a"
        return "yes"
```

Figure 1: A recognizer for a^+

In this example, compiler a1 corresponds to the second alternative of the production whose lhs is nonterminal A. Compiler a tries the first alternative and, if parsing fails, the second alternative is tried, by calling a1. The names of each compiler are generated from the name of the nonterminal at the lhs. 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. Alternatives after the first have parser names with a suffix that is a number, from 1 upwards, according to their textual order.

A *lexeme* is a grammar symbol that may be followed by *white spaces* (i.e. either blanks, newlines or tabs). The parser for a terminal symbol s is one of the lexical analyzers string s or symb s, depending on whether s is a lexeme or not, respectively. The following conventions is used:

By default, every nonterminal and terminal symbol is not a lexeme. The user can, however, either explicitly specify a lexemes section, for listing all (nonterminal or terminal) lexemes. A flag can also be used to allow the following *lexeme convention* for distinguishing nonterminals and terminals as lexemes or not:

- if the name of a nonterminal symbol starts with a lowercase letter, then it is a lexeme, otherwise it is not. For example, in the 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.

In this paper, we write input grammars assuming the lexeme convention.

Mímico allows predefined parsers to be imported, by means of an import clause. For example, '`import Predef (spaces);`' (or simply '`import Predef;`') enables the use of `spaces` to parse spaces.

Parsers generated for alternative productions of the same nonterminal symbol are combined by using the nondeterministic choice parser combinator, denoted by `<|>` (see, for example, [2, 14]). `p <|> q` concatenates the results of `p` and `q` into a list, considering *failure* as an empty list of successes [16] (in other words, when applied to `s`, parser `p <|> q` returns the list of results of `p` applied to `s` followed by the list of results of `q` applied to `s`).

Nondeterministic parsing, which occurs due to the use of the nondeterministic choice combinator (`<|>`), allows as input grammars that are neither LL(k) nor LR(k), for any k. We illustrate parsing of such a grammar with a simple example at the end of this section.

`apply p` is defined as

```
run (p >>= \x -> eoi >> return x)
```

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

The semantic rules can be used for more interesting purposes, of course, other than merely specifying a (kind of) language recognizer. The simple example in Figure 2 shows the input grammar and generated program for counting the number of `a`'s in an input sequence (import clauses are omitted hereafter, for brevity).

Compilers `a` and `a1` correspond to the first and second alternative productions of nonterminal `A`, respectively, and `a_1` is used to receive the result of `a`'s compilation.

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

- if a nonterminal `v` occurs more than once in a rhs, each occurrence of `v` must be distinguished in the semantic rule by a subscript, numbered from 1 upwards, and interpreted as denoting the nonterminal occurrence according to the textual order in the rhs.

  For example, production

  $$\mathsf{e} = \mathsf{e} + \mathsf{e}\ \{|\mathsf{e\_1} + \mathsf{e\_2}|\}$$

  specifies `e_1` as the result of the compilation of the first occurrence of `e` in the rhs, and `e_2` as the result of the compilation of the second occurrence of `e`.

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

Input grammar

```
compile = apply a

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

a1 = do string "a"
       return 1
```

Generated output

Figure 2: Counting the number of a's

- the absence of a semantic rule is considered as a semantic rule textually identical to the rhs.

  In this case, 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 subscripts indicate the order of occurrence in the rhs.

To illustrate the importance of testing the end of input in the additional production generated by Mímico, and the influence of the Pastor rule in the efficiency of generated parsers, consider now the following simple input grammar:

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

Given input `aa`, the program generated by Mímico for this grammar will, initially, successfully parse a single `a`. The additional production generated by Mímico then fails, because `eoi` fails. The parser tries then the second alternative, obtaining, lazily, the next element in a list of successful parser results. Both `a`'s are now consumed, and `eoi` thus succeeds.

Without the automatically generated parser for testing the end of input, the above grammar would

```
compile = apply p

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

p1 = do char_1 <- sat (const True)
        return True
     <|> return True
```

Figure 3: Program for recognizing palindromes

successfully parse *all* the input only if this input consisted of a single symbol a. The above input leads (due to following the Pastor rule) to a less efficient parsing than that of our first example, where the order of the alternatives is reversed.

A final example in this section illustrates the use of nondeterministic parsing to allow parsing of infinite look-ahead grammars that are neither LL(k) nor LR(k), for any k. The example is of a simple grammar for recognizing palindromes:

```
 P   = Char P Char   {| Char_1 == Char_2 |}
     | Char          {| True |}
     | epsilon       {| True |}
```

Char and epsilon are reserved nonterminals, representing a single arbitrary terminal symbol and an empty sequence of grammar symbols, respectively. The simple program generated by Mímico is shown in Figure 3.

The use of the nondeterministic choice operator <|> allows this simple grammar for palindromes to be given as input to Mímico. Consider as an example the parsing of input string aa, which 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 succeeds, returning lazily the list of successful parses [("a",""),("","a")]. Of these, the first does not originate a successful parse for the first call to p1; but the second does, after a backtrack, in which no input is consumed on the recursive call.

Next section formalizes the syntax and semantics of productions. Section 3 describes Mímico's handling of left recursion. Section 4 describes the use of monadic code in semantic rules and Section 5 the treatment of precedence and associativity of binary infix operators. Section 6 concludes.

2 Syntax and Semantics

We use the following meta-variables and corresponding syntactic domains:

Meta-var.	Domain	Name
G		grammar
$\mathbb{P}$		list of productions
P		production
$\mathtt{A}, \mathtt{B}, \mathtt{L}, \mathtt{R}, \mathtt{X}, \mathtt{Y}$	$\mathbb{V}$	nonterminal
$\mathtt{S}$	$\mathbb{V}$	initial nonterminal
a, b	Σ	terminal
u, v, w	Σ^*	sequence of terminals
r	$(\Sigma \cup \mathbb{V})^*$	sequence of grammar symbols
epsilon	$(\Sigma \cup \mathbb{V})^*$	empty sequence of grammar symbols
ϵ		empty sequence (the context determines the type of elements)
e, *code*		Haskell expression

The syntax of Mímico's input grammars is given below, where meta-symbols are written in a gray box (as in $\boxed{\texttt{::=}}$ and $\boxed{|}$), to avoid confusion. We do not include here the possibility of specifying the associativity and precedence of binary infix operators, which is presented in Section 5.

$$\begin{array}{lll} G & \boxed{::=} & P \;\boxed{|}\; P\,P' \\ P & \boxed{::=} & A = r_1 \;\{\!|\; e_1 \;|\!\}\;\; | \;\; \boxed{\cdots} \;\; | \;\; r_n \;\{\!|\; e_n \;|\!\} \end{array}$$

The result of processing an input string by a given grammar $G = (\Sigma, \mathbb{V}, S, \mathbb{P})$ is defined by a proof system that defines the meanings of productions and nonterminals of this grammar, with respect to a given input string. The axioms of the proof system are defined in Figure 4. The proof system includes also inference rules (omitted for brevity) that define provable equality as a congruence (i.e. an equivalence relation, preserved by substituting "equals for equals" — i.e. by substituting x by x' if $x = x'$ is provable).

$\mathbb{P}(\mathtt{A})$ denotes the list of rhs of alternatives for nonterminal A in G, and $a \bullet u$ denotes the sequence au (with head a and tail u). Subscripts G are written explicitly in the semantic functions, since we compare meanings given by distinct grammars in Section 3.

$\mathit{parsed}_G(u, r)$, used in this Figure, gives a prefix of u generated from r by successively replacing nonterminals by rhs of productions in G. parsed_G is defined in Figure 5.

$$[\![A]\!]_{G,u} \;=\; [\![A = r_1\,\{\!|\,e_1\,|\!\}\mid\;\cdots\;\mid r_n\,\{\!|\,e_n\,|\!\}]\!]_{G,u} \quad (1)$$

where $r_1\,\{\!|\,e_1\,|\!\}\mid\;\cdots\;\mid r_n\,\{\!|\,e_n\,|\!\} = \mathbb{P}(\mathtt{A})$

$$[\![A = r\{\!|e|\!\}]\!]_{G,u} \;=\; e[[\![A_1]\!]_{G,w_1}/A_1,\ldots,[\![A_n]\!]_{G,w_n}/A_n] \quad (2)$$

$$\begin{aligned}\text{where}\quad & r = u_1A_1u_2A_2\cdots u_nA_nu_{n+1}\\ & u_i \in \Sigma^*, \text{for } i = 1..n+1, \text{ and}\\ & \mathit{parsed}_G(u,A) = u_1w_1u_2w_2\cdots u_nw_nu_{n+1}\\ & w_i \in \Sigma^*, \text{for } i = 1..n\end{aligned}$$

$$[\![A = r_1\,\{\!|\,e_1\,|\!\}\mid\;\cdots\;\mid r_n\,\{\!|\,e_n\,|\!\}]\!]_{G,u} \;=\; \begin{cases} [\![A = r_1\,\{\!|\,e_1\,|\!\}]\!]_{G,u} & \text{if } p(u,r_1)\\ [\![A = r_2\,\{\!|\,e_2\,|\!\}]\!]_{G,u} & \text{if not } p(u,r_1) \text{ and } p(u,r_2)\\ \ldots & \\ [\![A = r_n\,\{\!|\,e_n\,|\!\}]\!]_{G,u} & \text{if not } p(u,r_1) \text{ and not } \ldots\\ & \ldots \text{not } p(u,r_{n-1})\\ & \quad\text{and } p(u,r_n)\\ \text{fail} & \text{if not } p(u,r_i),\\ & \quad\text{for all } i = 1..n \end{cases} \quad (3)$$

where $p(u,r) = (\mathit{parsed}_G(u,r) \neq \text{fail})$

Figure 4: Semantics of Mímico's input

3 Left Recursion

Since monadic parsing is based on a recursive descent technique, one might expect that left-recursive productions would not be 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 argument that left-recursive grammars can be rewritten to an equivalent non-left-recursive grammar.

However, a non-left-recursive grammar may not be so simple to specify as its left-recursive counterpart, specially with respect to semantic rules. Moreover, the semantic rules of the non-left-recursive grammar may turn out to specify a less efficient algorithm.

Mímico allows left-recursive grammars as input, requiring though that they have a certain simple form, as described below, in order that semantic rules can still be used. Left-recursive productions can appear in two forms, defined respectively in this section and in Section 5.

Let us consider first a simple example of a left-recursive grammar, describing the language {ab*}:

$$\begin{array}{lll} \mathit{parsed}_G(\epsilon, r) & = \epsilon & \\ \mathit{parsed}_G(u, \epsilon) & = \epsilon & \\ \mathit{parsed}_G(au, ar) & = a \bullet \mathit{parsed}_G(u,r) & \\ \mathit{parsed}_G(au, br) & = \text{fail (for } a \neq b) & \\ \mathit{parsed}_G(u, \mathtt{A}r) & = & \\ \quad \textbf{let}\ \ r_1\,\{\!|\,e_1\,|\!\}\mid\;\cdots\;\mid r_n\,\{\!|\,e_n\,|\!\} = \mathbb{P}(\mathtt{A}) & & \\ \quad \textbf{in}\ \ \mathit{parsed}_G(u, r_i), \text{ where } 1 \leq i \leq n \text{ is} & & \\ \qquad\quad \text{the least value such that} & & \\ \qquad\quad \mathit{parsed}_G(u,r_i) \text{ does not fail,} & & \\ \qquad \text{if it exists; otherwise fail} & & \end{array}$$

Figure 5: *parsed* substring

```
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 grammar, Mímico generates the program shown in Figure 6.

```
compile = apply x

x = do b_1 <- b
       x'_1 <- x'
       return (let f z x = z + 1
               in foldl f b_1 x'_1)

b = do string "a"
       return 0

x' = do undef_1 <- undef
        x'_1 <- x'
        return ( undef_1 :  x'_1 )
     <|> return []

undef = do string "b"
           return undefined
```

Figure 6: Program for counting b's in {ab*}

Left-recursion is treated by using essentially the standard left-recursion elimination algorithm for context-free grammars [1]. A novel aspect of our treatment of left-recursive productions is the use of an intermediate list to handle the results of the parser obtained by the transformation of a left recursive grammar. This is needed because we cannot split productions, so that we can generate code for the original semantic rules. In the particular case of the program above, a list of undefined values is constructed while

```
$first_G(r)$           = $first_G(r, \emptyset)$
$first_G(\epsilon, \mathbb{S})$        = $\{\epsilon\} \cup \mathbb{S}$
$first_G(au, \mathbb{S})$      = $\{a\} \cup \mathbb{S}$
$first_G(Ar, \mathbb{S})$      =
    if  A $\in \mathbb{S}$ then $\mathbb{S}$
    else let $A = r_1 \{| e_1 |\} \mid \cdots \mid r_n \{| e_n |\} \in \mathbb{P}(\mathtt{A})$
         in  $\{\mathtt{A}\} \cup \mathbb{S} \cup \mathbb{S}' \cup \bigcup_{i=1}^{n} first_G(r_i, \{\mathtt{A}\} \cup \mathbb{S})$
           where $\mathbb{S}'$ = if  $\epsilon \in \mathbb{S}$ or $r_i = \epsilon$,
                                  for some $i \in \{1, \ldots, n\}$
                              then  $first_G(r)$
                              else  $\emptyset$
```

Figure 7: Function *first*$_G$

```
A  =  A R$_1$   $\{| e_1 |\}$
   |  ...
   |  A R$_n$   $\{| e_n |\}$
   |  $r_1$     $\{| e'_1 |\}$
   |  ...
   |  $r_m$     $\{| e'_m |\}$
```

where, for $j = 1, \ldots, m$, r_j is not left-recursive and, for $i = 1, \ldots, n$, R_i is restricted to denote a single nonterminal symbol or a sequence of terminal symbols (i.e. $\mathtt{R}_i \in \mathbb{V} - \{\mathtt{A}\}$ or $\mathtt{R}_i = w$, where $w \in \Sigma^+$).

Figure 8: Form of directly left-recursive productions

each input character b is consumed. Then, folding the constructed list with the function generated from the semantic rule gives the desired result (in this case, the number of elements of the list).

The general scheme of transformation of left-recursive productions is explained next. We consider first direct left-recursive productions, and then generalize the translation scheme to handle indirect left-recursion.

For any sequence of grammar symbols r, *first*$_G(r)$ gives the set of first (terminal or nonterminal) symbols that can be generated from r, by successively replacing a nonterminal occurring in r by the rhs of a production in G for that nonterminal. Function *first*$_G$ is defined in Figure 7.

A production $\mathtt{A} = r \{| e |\}$ is *left-recursive* if $\mathtt{A} \in$ *first*$_G(r)$. A nonterminal is *left-recursive* if it has a left-recursive production.

The scheme of eliminating left-recursive productions in this section requires left-recursive productions to have the form shown in Figure 8, apart from the possibility of handling indirect left recursion, discussed at the end of this section, and from the fact that left-recursive alternatives need not appear all before non-left-recursive ones, as shown in Figure 8 (this form is chosen for the sake of readability).

The restriction to this form is used to enforce that semantic rule e_i be written with respect to A and R$_i$ (if $\mathtt{R}_i \in \mathbb{V}$), for $i = 1, \ldots, n$ (see the scheme of elimination of left-recursive productions in Figure 9). It is significantly easier, though, to transform left-recursive productions into this form, than to eliminate left-recursion altogether (this is illustrated below by an example that specifies the transformation of integer literals from binary to decimal notation).

Figure 9 shows the transformed grammar, obtained from the left-recursive productions in the form of Figure 8.

A$'$, B and Y$_i$, for each $i = 1, \ldots, n$, are fresh nonterminal symbols. Letting $e[e'/\mathtt{A}]$ denote (as usual) the string obtained from e by substituting e' for each

```
A      =  B A$'_1$       $\{| code_1 |\}$
       |  ...
       |  B A$'_n$       $\{| code_n |\}$

B      =  $r_1$          $\{| e'_1 |\}$
       |  ...
       |  $r_m$          $\{| e'_m |\}$

A$'_1$     =  Y$_1$ A$'$       $\{|$ Y$_1$ : A$'$ $|\}$
       |  epsilon    $\{|$ [] $|\}$
...
A$'_n$     =  Y$_n$ A$'$       $\{|$ Y$_n$ : A$'$ $|\}$
       |  epsilon    $\{|$ [] $|\}$

Y$_1$     =  R$_1$         $\{|$ X$_1$ $|\}$
...
Y$_n$     =  R$_n$         $\{|$ X$_n$ $|\}$
for $i = 1, \ldots, n$    $\mathtt{X}_i = \begin{cases} \mathtt{R}_i & \text{if } \mathtt{R}_i \in \mathbb{V} \\ \mathtt{undefined} & \text{otherwise} \end{cases}$

A$'$      =  A$'_1$         $\{|$ A$'_1$ $|\}$
       |  ...
       |  A$'_n$         $\{|$ A$'_n$ $|\}$
```

Figure 9: Direct left-recursion elimination

occurrence of A in e, we have, for $i = 1, \ldots, n$:

$$code_i = \begin{cases} \texttt{foldl}\ f_i\ \texttt{B}\ \texttt{A}'\ \text{where}\ f_i\,\texttt{z x = } e_i\texttt{[z/A, x/R}_i\texttt{]} \\ \qquad \text{if } \texttt{R}_i \in \mathbb{V} \\ \texttt{foldl}\ f_i\ \texttt{B}\ \texttt{A}'\ \text{where}\ f_i\ \ \texttt{z x = } e_i\texttt{[z/A]} \\ \qquad \text{otherwise} \end{cases}$$

where we assume that x and z are fresh variables (i.e. they do not occur in any e_i, for $i = 1, \ldots, n$).

The following also holds: i) if, for $1 \geq j \geq m$, $\texttt{r}_j$ is a single nonterminal, then production B = r_j is not needed; ii) when $\texttt{R}_i$ is a single nonterminal, production $\texttt{Y}_i$ = $\texttt{R}_i$ is not needed and every occurrence of $\texttt{Y}_i$ can be replace by $\texttt{R}_i$ (for $i = 1, \ldots, n$).

The following 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
  | 1
```

Left recursion not only simplifies the semantic rule in this case; it also leads to the generation of more efficient code. Compare with the following (right-recursive) alternative:

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

B = 0
  | 1
```

Based on the left-recursion elimination scheme, Mímico generates, for the left-recursive input grammar, the program shown in Figure 10.

We now prove the following:

Theorem 1 Let G_0 be a left-recursive grammar in the form required by the transformation scheme above, and let G_1 be the grammar obtained as a result of applying the left-recursion elimination scheme. For any nonterminal A of G_0,

$$[\![\texttt{A}]\!]_{G_0,u} = [\![\texttt{A}]\!]_{G_1,u}$$

```
compile = apply l

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

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

b = do string "0"
       return 0
    <|> b1

b1 = do string "1"
        return 1
```

Figure 10: Generated binary to decimal converter

Proof: Since G_1 only alters G_0 with respect to the transformation of left-recursive productions, we consider in the proof only left-recursive nonterminals in G_0. We prove only the case for $n = m = 1$. The generalization for $n > 1$ and $m > 1$ can be obtained straightforwardly by induction on n and m, respectively (note that, because of the use of the Pastor rule, we need not consider problems of ambiguity, that would otherwise arise when the set of first symbols does not determine the alternative to be used). We omit the subscripts in $\texttt{R}_1, r_1, e_1, e'_1, \texttt{A}'_1, code_1, \texttt{Y}_1$ and $\texttt{X}_1$ and consider, for brevity, that $\texttt{R} \in \mathbb{V}$, since the case for $\texttt{R} = w, w \in \Sigma^*$, is similar. We also assume that $[\![\texttt{A} = r\,\{|\,e'\,|\}]\!]_{G_0,u} = [\![\texttt{B}]\!]_{G_1,u}$ and $[\![\texttt{R}]\!]_{G_0,u} = [\![\texttt{R}]\!]_{G_1,u}$, for any $u \in \Sigma^*$ (since G_1 only differs from G_0 by the transformation of left-recursive productions), and use the following straightforward lemma (which can be seen as an alternative definition of foldl, that would lead, though, with the usual implementation of lists, to an inefficient implementation):

Lemma 1 If x is not null then foldl f z x is equal to f (last x) (foldl f z (init x)), otherwise z.

Let $\mathbb{B} = [\![\texttt{A} = r\{|e'|\}]\!]_{G_0,u}$, for brevity, and consider:

- $[\![\texttt{A}]\!]_{G_0,u} =$ fail. Then $[\![\texttt{A}]\!]_{G_1,u} =$ fail.
- u is such that $parsed_{G_0}(u, \texttt{A R}) =$ fail and $parsed_{G_0}(u, r) = u$.

$$\begin{aligned}\text{Then}\quad \llbracket \mathtt{A}\rrbracket_{G_0,u} &= \llbracket \mathtt{A} = \mathtt{A\,R}\,\{\!|\, e \,|\!\}\ \mid\ r\ \{\!|\, e' \,|\!\}\rrbracket_{G_0,u}\\ &= \llbracket \mathtt{A} = r\ \{\!|\, e' \,|\!\}\rrbracket_{G_0,u} = \mathbb{B}\end{aligned}$$

and: $\llbracket \mathtt{A}\rrbracket_{G_1,u} = \llbracket\ \mathtt{A\ =\ B\ A'}\ \{\!|\ \mathit{code}\ |\!\}\rrbracket_{G_1,u}$ where

$$\begin{aligned}\mathit{code} &= \mathtt{foldl\ f}\ (\llbracket \mathtt{B}\rrbracket_{G_1,u})\ (\llbracket \mathtt{A}'\rrbracket_{G_1,\epsilon})\\ \mathtt{f\ z\ x} &= e[\mathtt{z}/\mathtt{A}, \mathtt{x}/\mathtt{R}]\end{aligned}$$

Since $\llbracket \mathtt{A}'\rrbracket_{G_1,\epsilon} = \mathtt{[]}$, we have that

$$\llbracket \mathtt{A}\rrbracket_{G_1,u} = \mathtt{foldl\ f}\ \mathbb{B}\ \mathtt{[]} = \mathbb{B}$$

- otherwise, let u be such that $\mathit{parsed}_{G_0}(u,\mathtt{A}) = u$. We have that u must be then of the form vw (for some $v, w \in \Sigma^*$) where $w = w_1 w_2 \ldots w_k$ is such that
$\mathit{parsed}_{G_0}(w_i,\mathtt{R}) = w_i$, for all $i = 1,\ldots,k$, $\mathit{parsed}_{G_0}(v,\mathtt{A\,R})$ fails and $\mathit{parsed}_{G_0}(v,r) = v$ (otherwise we would have $\llbracket \mathtt{A}\rrbracket_{G_0,u} = \llbracket \mathtt{A}\rrbracket_{G_1,u} = \text{fail}$). For all $i = 1,\ldots,k$, $\mathit{parsed}_{G_1}(w_i,\mathtt{R}) = w_i$. Let also $\mathbb{B} = \llbracket \mathtt{B}\rrbracket_{G_1,v}$ and $\mathbb{R}_i = \llbracket \mathtt{R}\rrbracket_{G_0,w_i}$, for $i = 1,\ldots,k$, as abbreviations. We proceed by induction on k:

  - Base case: $k = 1$. Then:

$$\begin{aligned}\llbracket \mathtt{A}\rrbracket_{G_0,vw} &= \llbracket \mathtt{A} = \mathtt{A\,R}\,\{\!|\, e \,|\!\}\ \mid\ r\ \{\!|\, e' \,|\!\}\rrbracket_{G_0,vw}\\ &= \llbracket \mathtt{A} = \mathtt{A\,R}\,\{\!|\, e \,|\!\}\rrbracket_{G_0,vw}\\ &= e\ [\ \llbracket \mathtt{A}\rrbracket_{G_0,v}/\mathtt{A}, \mathbb{R}_1/\mathtt{R}\ ]\end{aligned}$$

Thus $\llbracket \mathtt{A}\rrbracket_{G_0,vw} = e[\mathbb{B}/\mathtt{A}, \mathbb{R}_1/\mathtt{R}]$. We have also that
$\llbracket \mathtt{A}\rrbracket_{G_1,vw} = \llbracket \mathtt{A} = \mathtt{B\,A}'\ \{\!|\ \mathit{code}\ |\!\}\rrbracket_{G_1,vw}$, where

$$\begin{aligned}\mathit{code} &= \mathtt{foldl\ f}\ (\llbracket \mathtt{B}\rrbracket_{G_1,v})\ (\llbracket \mathtt{A}'\rrbracket_{G_1,w})\\ \mathtt{f\ z\ x} &= e[\mathtt{z}/\mathtt{A}, \mathtt{x}/\mathtt{R}]\end{aligned}$$

Now:

$$\begin{aligned}\llbracket \mathtt{A}'\rrbracket_{G_1,w} &= \llbracket \mathtt{A}' = \mathtt{Y\,A}'\ \{\!|\ \mathtt{Y : A}'\ |\!\}\rrbracket_{G_1,w}\\ &= \mathtt{Y : A}'\ [\ \llbracket \mathtt{Y}\rrbracket_{G_1,w}/\mathtt{Y},\ \llbracket \mathtt{A}'\rrbracket_{G_1,\epsilon}/\mathtt{A}'\ ]\\ &= \llbracket \mathtt{Y}\rrbracket_{G_1,w}\ \mathtt{:}\ \ \mathtt{[]}\\ &= [\ \mathbb{R}_1\ ]\\ \llbracket \mathtt{A}'\rrbracket_{G_1,\epsilon} &= \llbracket \mathtt{A}' \mathtt{=\ epsilon\ \{|\ []\ |\}}\rrbracket_{G_1,\epsilon}\\ &= \mathtt{[]}\end{aligned}$$

and thus

$$\begin{aligned}\llbracket \mathtt{A}\rrbracket_{G_1,wv} &= \mathtt{foldl\ f}\ (\llbracket \mathtt{B}\rrbracket_{G_1,v})\ [\ \mathbb{R}_1\ ]\\ &= \mathtt{foldl\ f}\ \mathtt{(f}\ \mathbb{B}\ \mathbb{R}_1\mathtt{)}\ \mathtt{[]}\\ &= \mathtt{f}\ \mathbb{B}\ \mathbb{R}_1\\ &= e[\mathbb{B}/\mathtt{A}, \mathbb{R}_1/\mathtt{R}]\end{aligned}$$

  - Inductive case. We have:

$$\begin{aligned}\llbracket \mathtt{A}\rrbracket_{G_0,vw} &= \llbracket \mathtt{A} = \mathtt{A\,R}\,\{\!|\, e \,|\!\}\ \mid\ r\ \{\!|\, e' \,|\!\}\rrbracket_{G_0,vw}\\ &= \llbracket \mathtt{A} = \mathtt{A\,R}\,\{\!|\, e \,|\!\}\rrbracket_{G_0,vw}\\ &= e[\ \llbracket \mathtt{A}\rrbracket_{G_0,vw_1\cdots w_{k-1}}/\mathtt{A}, \llbracket \mathtt{R}\rrbracket_{G_0,w_k}/\mathtt{R}\ ]\\ &= e[\ \llbracket \mathtt{A}\rrbracket_{G_0,vw_1\cdots w_{k-1}}/\mathtt{A}, \mathbb{R}_k/\mathtt{R}\ ]\end{aligned}$$

$$\begin{aligned}\text{and}\quad \llbracket \mathtt{A}\rrbracket_{G_1,vw} &= \llbracket \mathtt{A} = \mathtt{B\,A}'\ \{\!|\ \mathit{code}\ |\!\}\rrbracket_{G_1,vw}\\ &= \mathit{code}\end{aligned}$$

$$\begin{aligned}\text{where:}\quad \mathit{code} &= \mathtt{foldl\ f}\ (\llbracket \mathtt{B}\rrbracket_{G_1,v})\ (\llbracket \mathtt{A}'\rrbracket_{G_1,w})\\ \mathtt{f\ z\ x} &= e[\mathtt{z}/\mathtt{A}, \mathtt{x}/\mathtt{R}]\end{aligned}$$

By the induction hypothesis,
$\llbracket \mathtt{A}\rrbracket_{G_0,vw_1\cdots w_{k-1}} = \llbracket \mathtt{A}\rrbracket_{G_1,vw_1\cdots w_{k-1}}$ and
$\llbracket \mathtt{R}\rrbracket_{G_0,w_k} = \llbracket \mathtt{R}\rrbracket_{G_1,w_k}$.
Thus:
$e[\ \llbracket \mathtt{A}\rrbracket_{G_0,vw_1\cdots w_{k-1}}/\mathtt{A}, \llbracket \mathtt{R}\rrbracket_{G_0,w_k}/\mathtt{R}\ ] =$
$e[\ \llbracket \mathtt{A}\rrbracket_{G_1,vw_1\cdots w_{k-1}}/\mathtt{A}, \llbracket \mathtt{R}\rrbracket_{G_1,w_k}/\mathtt{R}\ ]$
Since $\llbracket \mathtt{A}\rrbracket_{G_1,vw_1\cdots w_{k-1}} =$
$\mathtt{foldl\ f}\ (\llbracket \mathtt{B}\rrbracket_{G_1,v})\ (\llbracket \mathtt{A}'\rrbracket_{G_1,w_1\cdots w_{k-1}})$
we obtain:

$$\begin{aligned}&\llbracket \mathtt{A}\rrbracket_{G_0,vw}\\ &= e[\ \mathtt{foldl\ f}\ (\llbracket \mathtt{B}\rrbracket_{G_1,v})\ (\llbracket \mathtt{A}'\rrbracket_{G_1,w_1\cdots w_{k-1}})/\mathtt{A},\\ &\qquad \llbracket \mathtt{R}\rrbracket_{G_1,w_k}/\mathtt{R}\ ]\\ &= \mathtt{f}\ \big(\mathtt{foldl\ f}\ (\llbracket \mathtt{B}\rrbracket_{G_1,v})\ (\llbracket \mathtt{A}'\rrbracket_{G_1,w_1\cdots w_{k-1}})\big)\ (\mathbb{R}_k)\end{aligned}$$

Then, by lemma 1, $\llbracket \mathtt{A}\rrbracket_{G_0,vw} = \llbracket \mathtt{A}\rrbracket_{G_1,vw}$

To extend the left-recursion elimination scheme to handle indirect left-recursion, we need to modify, in Figure 8, the rhs of productions which start with nonterminal $\mathtt{A}$, in a very simple way, to allow nonterminals to be not only directly but also indirectly left-recursive. The form of left-recursive productions becomes the one shown in Figure 11.

$$\boxed{\begin{array}{lcll}\mathtt{A} & = & \mathtt{L}_1\,\mathtt{R}_1 & \{\!|\, e_1 \,|\!\}\\ & \mid & \ldots & \\ & \mid & \mathtt{L}_n\,\mathtt{R}_n & \{\!|\, e_n \,|\!\}\\ & \mid & r_1 & \{\!|\, e'_1 \,|\!\}\\ & \mid & \ldots & \\ & \mid & r_m & \{\!|\, e'_m \,|\!\}\end{array}}$$

where, for $j = 1,\ldots,m$, r_j is not left-recursive and, for $i = 1,\ldots,n$, L_i is left-recursive and R_i is restricted to denote a single nonterminal symbol or a sequence of terminal symbols (i.e. $\mathtt{R}_i \in \mathbb{V} - \{\mathtt{A}\}$ or $\mathtt{R}_i = w$, where $w \in \Sigma^+$).

Figure 11: Left-recursive productions (required form)

$$\begin{array}{lll}
\mathtt{B} & = & r'_1 \quad \{\!|\, e''_1 \,|\!\} \\
 & | & \ldots \\
 & | & r'_k \quad \{\!|\, e''_k \,|\!\} \\
\end{array}$$

where $(r'_1, e''_1), \ldots, (r'_k, e''_k) = \mathit{nlr}_G(\mathtt{A})$

Figure 12: Handling indirect left-recursion

Figure 9 is modified only with respect to the productions for B, which are shown in Figure 12 and are given by function nlr_G. $\mathit{nlr}_G(\mathtt{A})$ gives a *list* of pairs of rhs and semantic rules, of productions of nonterminals in $\mathit{first}_G(\mathtt{A})$ for which the rhs is not left-recursive:

$$\begin{array}{l}
\mathit{nlr}_G(A) \quad = \mathit{nlr}_G(A, \emptyset) \\
\mathit{nlr}_G(A,\mathbb{S}) = \textbf{if } \mathtt{A} \in \mathbb{S} \textbf{ then } \ \epsilon \ \textbf{ else } s_1 \circ s_2 \circ \ldots s_n \\
\text{where } r_1 \{\!|\, e_1 \,|\!\} \mid \cdots \mid r_n \{\!|\, e_n \,|\!\} = \mathbb{P}(\mathtt{A}) \\
\qquad \circ \text{ denotes sequence concatenation} \\
\qquad \text{for } i = 1..n, \\
\qquad s_i = \begin{cases} \epsilon & \text{if } r_i = \mathtt{B}\, r, \text{ for some } \mathtt{B}, r \\ & \text{s.t. } \mathtt{B} \in \mathbb{S} \\ \mathit{nlr}_G(\mathtt{B}, \mathbb{S} \cup \mathtt{A}) & \text{if } r_i = \mathtt{B}\, r, \text{ for some } \mathtt{B}, r \\ & \text{s.t. } \mathtt{B} \notin \mathbb{S}, \mathtt{A} \in \mathit{first}_G(\mathtt{B}) \\ (r_i, e_i) & \text{otherwise} \end{cases}
\end{array}$$

4 Monadic code

Two distinct approaches are provided for using monadic code in semantic rules, explained in the next subsections. The *general* approach can be used in a wide range of situations and is the default process adopted by Mímico. The *strict parsing* approach allows semantic rules to use context-sensitive conditions to control the parsing process. It is assumed upon the use of a flag when starting Mímico.

4.1 General approach

The general approach constructs a chain of user defined monadic semantic rules, with the monadic bind operator. The monadic code is not executed immediately after a successful parse of the corresponding production; it is returned, instead, as the overall parser result.

The usual substitutions and adaptations in the semantic rule are performed, to allow references in the monadic semantic rule to the production right hand side.

```
Module = DecL ExpL    {| () |}
DecL   = Dec DecL     {| () |}
       | Dec
Dec    = id :  type   [| id 'hasType' type |]
ExpL   = Exp ExpL     {| () |}
       | Exp
Exp    = id = value   [|

    typeOf value >>=  \tvalue ->
    typeOf ident >>=  \tident ->
    if tvalue 'isCompatibleWith' tident
       then ident 'hasValue' value
       else error "Wrong type" |]
```

Figure 13: Grammar sketch for 'define-before-use'

It is possible for some semantic rules to be monadic and others not. To properly sequence monadic actions, every non-monadic action that has a non-terminal with a monadic semantic rule in its rhs is promoted to monadic. For example, consider Figure 13, where part of a tiny imperative language is defined, where an identifier must be declared before used. Definitions of *ident*, *value*, *type* and expressions other than assignment are omitted for brevity.

A semantic rule between '[|' and '|]' is assumed to contain monadic code. Semantic rules of productions depending on productions having monadic rules must be promoted to monadic ones. This imposes a slight overhead in Mímico's generated parsers, because only the directly or indirectly dependent semantic rules have to be handled as monadic.

The implementation of the general approach divides nonterminals in two groups, called monadic and non-monadic. The monadic group M is defined inductively by the subset of nonterminals whose productions have a monadic semantic rule, or nonterminals which have a rhs in which there is a nonterminal in M. The non-monadic group is M's complement. For example, in the grammar of Figure 13, the monadic group contains the nonterminals in $\{\mathtt{Decl}, \mathtt{Exp}, \mathtt{DeclL}, \mathtt{Module}, \mathtt{ExpL}\}$.

The generated parser is shown in Figure 14.

In the general approach, the introduction of a monadic semantic rule in a production for A changes the code of parsers generated for other nonterminals if and only if they have a production whose rhs uses A. Thus, the generated program does not change due to the introduction of a few "localized" monadic semantic rules. Also, the monad defined by the user is completely independent from Mímico's monad. Mímico needs to know only the bind and return operators of the user monad. The definitions of these operators can be supplied in the input grammar by a special directive. If no such directive is defined, the user monad is assumed to be an instance of the `Monad` class, and `>>=` and `return` are used. Finally, this approach works well in the presence of nondeterminism. Since monadic actions are only performed at the end of the parsing process, backtracking does not cause any problem or inconsistency in the state of the user's monad, even if the semantic rules specify `IO` operations.

```
compile = apply module

module =
   do mdecL_1 <- decL
      mexpL_1 <- expL
      return (mdecL_1 >>= \dec_1 ->
              mexpL_1 >>= \expL_1 ->
              return ())

decL  =
   do mdec_1 <- dec
      mdecL_1 <- decL
      return (mdec_1 >>= \dec_1 ->
              mdecL_1 >>= \decL_1 ->
              return ())
   <|> decL1

decL1 =
   do mdec_1 <- dec
      return (mdec_1 >>= \dec_1 ->
              return ())

dec   =
   do ident_1 <- ident
      token ":"
      type_1 <- type
      return (ident_1 ’hasType’ type_1)

expL  =       ...
...further code omitted for brevity ...
```

Figure 14: Compiler for tiny language

As a drawback, semantic rules cannot be used to control the parsing process (cf. e.g. [13, 12]). This is enabled by the strict parsing approach described in the following section.

4.2 Strict parsing approach

The strict parsing approach allows the use of monadic semantic rules to control the parsing process, allowing context-sensitive conditions to be checked during parsing.

The *fail* primitive can be used in semantic rules to stop parsing for the current alternative and start parsing of the next alternative, if it exists, otherwise causing failure.

Consider for example the distinction between function calls and array references in Fortran, which can both be denoted by an identifier followed by arguments between left and right parentheses. The grammar fragment shown in Figure 15 illustrates Mímico's way of handling context-sensitivity.

```
Exp   = Array ( ExpL ) {| ...|}
      | Func ( ExpL )  {| ...|}
Array = ident [| isArray ident >>= \is ->
                 unless is fail >>
                 return ident |]
Func  = ident [| isFunc ident >>= \is ->
                 unless is fail >>
                 return ident |]
```

Figure 15: Grammar fragment for function calls and array references in Fortran

This grammar fragment is translated to the program fragment shown in Figure 16.

In the strict parsing approach, grammars must be written carefully in order to avoid inconsistencies, due to the incompatibility of strict with nondeterministic parsing. As an example, a parser using IO operations could produce a strange output in the presence of backtracking.

To provide isolation from Mímico's internal state, the user monadic actions must be defined by means of the predefined set of operations `read`, `write`, and `trans`, used to read, write and transform the user defined state, respectively.

The definition used in Mímico is customized according to the definitions of bind and return provided by the user. As explained before, Mímico does not require the user state transformer to be an instance of class `Monad`; it only requires an explicit indication of which functions must be used as bind and return functions in the generated code.

```
exp   = do  array_1 <- array
            token '('
            expL_1 <- expL
            token ')'
            return ( ...)
      <|> exp1

exp1  = do  func_1 <- func
            token '('
            expL_1 <- expL
            token ')'
            return ( ...)

array = do  ident_1 <- ident
            (isArray ident_1 >>=
               \is-> unless is fail >>
                  return ident_1)

            fundident_1 <- ident
               (isFunc ident_1 >>=
                  \is-> unless is fail >>
                     return ident_1)
```

Figure 16: Strict parsing and the use of context sensitive conditions for parsing control

5 Precedence and Associativity

This section considers Mímico's handling of binary infix operators. Consider, for example, the archetypical left-recursive grammar for arithmetic expressions shown below (we consider only operators +, - and *, for brevity).

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

n is the name of a parser defined in Parser. It parses an integer numeral lexeme, returning the corresponding value. Note the abscence of semantic rules (as already mentioned, when a semantic rule is not explicitly specified, Mímico considers the production's rhs as the semantic rule).

This input grammar is not in the form required by the left-recursion elimination scheme of the previous section. However, Mímico will accept grammars in this form as input. Informally, the left-recursive alternatives must have, between a left-recursive nonterminal and another nonterminal, a single terminal or nonterminal — the latter required to consist of alternatives consisting of a single terminal. Formally, left-recursive productions in this section are required to have the form shown in Figure 17.

$$
\begin{array}{lcll}
\mathtt{A} & = & \mathtt{L}_1\ \mathtt{op}_1\ \mathtt{A}_1 & \{\!|\, e_1 \,|\!\} \\
 & | & \ldots & \\
 & | & \mathtt{L}_n\ \mathtt{op}_n\ \mathtt{A}_n & \{\!|\, e_n \,|\!\} \\
 & | & r_1 & \{\!|\, e'_1 \,|\!\} \\
 & | & \ldots & \\
 & | & r_m & \{\!|\, e'_m \,|\!\}
\end{array}
$$

where, for $i = 1, \ldots, n$, $\mathtt{L}_i$ is left-recursive, $\mathtt{op}_i \in \Sigma^*$ or $\mathtt{op}_i = r_{i1}\ \{\!|\, e_{i1} \,|\!\}\ |\ \cdots\ |\ r_{ip_i}\ \{\!|\, e_{ip_i} \,|\!\} \in \mathbb{P}$, where $r_{ij} \in \Sigma^*$, for all $j = 1, \ldots, p_i$, $\mathtt{A}_i \in \mathbb{V}$ and, for $j = 1, \ldots, m$, r_j is not left-recursive.

Figure 17: Required form of left-recursive productions with binary infix operators

Given the simplified grammar for arithmetic expressions above, Mímico generates as output the expression evaluator shown in Figure 18. We will look at this program shortly; before doing so, let us examine a simpler expression evaluator program, generated for the following similar input:

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

The dots inserted in this grammar specification indicate right-associativity of binary operators. This grammar specifies also, according to the *Pastor rule*, that + has the lowest precedence, then - and finally *. The program generated is shown in Figure 19. Parser token is such that token p applies parser p and then just removes trailing spaces; zerop is a parser that fails on any input.

The little compilers e and e1 to e4 correspond to alternative productions of nonterminal e. Compilers e, e1 and e2 are parameterized, to control operator precedence. The parameter denotes 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: a parameterized parser performs the longest possible parsing of the input not containing a string that is a member of the list of strings specified as arguments for this parser.

For example, parsers generated in the program of Figure 19 consider the binary infix operators of the input grammar as right-associative; the result of `compile "2 - 3 - 4"`, for example, is `3`, since the parameter in the call to `e2` in the body of `e1` contains `"-"`, and the longest possible parsing of `e2` not containing `"-"` is `2`. Symbol `"-"` is then consumed after this parsing of `e2`, and `e` parses the rest of the input.

Left-associativity, the default, is based on a simple extension of this scheme, based on *left factoring*. This is illustrated by the program shown in Figure 18. The first operand of a binary infix operator is simply saved until the second operand is obtained by using again the parameterization scheme.

It is possible to "override" the *Pastor rule* and specify that infix operators should have the same precedence. In this case, their associativity is the same as that of the symbol(s) with which they have the same precedence. For example:

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

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

Symbol `|=` means "same precedence, with respect to the operator in the previous alternative". The grammars on the left and right specify left and right associativity, respectively.

The programs generated in these cases (omitted for space reasons) differ only slightly from the corresponding ones above (for left and right associativity, respectively). 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. Similar programs are generated if a nonterminal is used in the place of infix operators.

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

rest s e_1 op0 =
      do e_2 <- token $ e1(["+"] ‘union‘ s)
         op <- if "+" ‘elem‘ s then zerop
               else do {symb "+"; return (+)}
         let opnd = op0 e_1 e_2
         rest s opnd op
      <|> do e_2 <- token $ e s
             return ( op0 e_1 e_2 )

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 $ e3(["-"] ‘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 $ e s
              return ( op0 e_1 e_2 )

e2 s = do e_1 <- token $ e3
          if"*" ‘elem‘ s then zerop
            else string "*"
          e_2 <- token $ e s
          return ( e_1 * e_2 )
       <|> e3

e3 = do n_1 <- token (some (sat isDigit))
        return ( read n_1 )
     <|> e4

e4 = do string "("
        e_1 <- token $ e []
        string ")"
        return e_1
```

Figure 18: Left factoring for left associativity

```
compile = apply (e[])

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

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

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

e3 = do n_1 <- token (some (sat isDigit))
        return ( read n_1 )
     <|> e4

e4 = do string "("
        e_1 <- token $ e []
        string ")"
        return e_1
```

Figure 19: Program for expression grammar with right associative binary infix operatorsx

6 Conclusion

We have described an approach to parser generation based on monadic combinators. A prototype is available at the URL

http://www.dcc.ufmg.br/~camarao/mimico

Work is going on in order to make the implementation more robust and user-friendly.

As far as we know, this is the first compiler generator based on monadic parsing. We have presented the general principle (the *Pastor rule*) and the ideas on which this work has been based: the scheme for supporting left-recursive input grammars, parameterization of compilers as a way of handling operator precedence and associativity, and the use of nondeterministic parsing to allow as input grammars that are neither LL(k) nor LR(k), for any k.

Among noticeable characteristics of Mímico are the clarity and readability of Mímico's input grammars and output programs and the fact that Mímico can handle input grammars that are neither LL(k) nor LR(k), for any k. For example, the simple grammar of palindromes presented in the introduction of this article would not be accepted by Yacc or Happy. The cost of this is, of course, efficiency, but that is related to Mímico's foundational work, monadic parsing, and in particular nondeterministic monadic parsing. Mímico's efficiency and further development depends thus on the efficiency and development of monadic parsing itself (see e.g. [14]).

Despite the readability of hand-written code of translators based on monadic parsing (in comparison to code produced by using other parsing techniques), the input for Mímico is significantly more readable, as can be seen by comparing Mímico's inputs with monadic parsers shown in this paper. Due to the readability of its input grammars, we expect Mímico to serve also as a useful tool for teaching formal specification of programming language syntax and semantics.

A lot of work has been done on the subject of compiler generators (see, for example, [10, 8, 11, 5]). We have not yet done performance comparisons with other compiler generators, like for example Yacc [10] and Happy [5], and discussions concerning the efficiency of generated compilers are left for further work (see e.g. [14, 9, 4] for discussions concerning the efficiency of monadic parsers and lexers). Our motivation up to now has centered on showing that it is possible and worthwhile to generate simple and readable monadic compilers automatically, from a still more readable and simple input grammar.

In the current implementation, if the input to Mímico or the input to the program generated by Mímico is syntactically wrong, the program simply halts, issuing a simple error message. A lot of work has to be done in order that parsers generated by Mímico provide good error reporting and recovery, which are fundamental tasks of a parser.

Further work includes the support of: i) concurrent parsing, with synchronization for termination of the concurrently initiated parsers when appropriate, ii) use of monadic code in the middle of productions, for passing information collected during parsing, through nonterminals, and iii) user controlled "local pruning", i.e. the option to control and avoid parser backtracking.

References

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

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

[3] Carlos Camarão and Lucília Figueiredo. A Monadic Combinator Compiler Compiler. In *Proc. SBLP'2001 (V Brazilian Symposium on Programming Languages)*, pages 64–79, 2001.

[4] M.T. Chakravarty. Lazy Lexing is Fast. In *Fourth Fuji International Symposium on Functional and Logic Programming (LNCS 1722, Springer-Verlag)*, 1999.

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

[6] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2:232–343, 1992.

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

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

[9] Daan Leijen. Parsec. http://www.cs.ruu.nl/˜daan/parsec.html.

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

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

[12] Terence Parr and Russell Quong. Adding semantic and syntactic predicates to ll(k): pred-ll(k). In *Proceeding of the International Conference on Compiler Construction*, 1994.

[13] Terence Parr and Russell Quong. Antlr: A predictive-ll(k) parser generator. *Software — Practice and Experience*, 25(7):789–810, 1995.

[14] S.D. Swierstra. Parser Combinators, from Toys to Tools. In *ACM SIGPLAN Haskell Workshop*, pages 113–128, 2000.

[15] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1999. second edition.

[16] Philip Wadler. How to replace a failure with a list of successes. In *Functional Programming Languages and Architecture, LNCS 201*, pages 113–128, 1985.

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

[18] Philip Wadler. Monads for Functional Programming. *Computer and Systems Sciences*, 118, 1992.