Parsing with Non-Deterministic Functions

Rafael Caballero Roldán, Francisco J. López Fraguas

Abstract

Parsing has been a traditional workbench for showing the virtues of declarative programming. Both logic and functional programming claim the ability of writing parsers in a natural and concise way. We address here the task from a functional-logic perspective. By modeling parsers as non-deterministic functions we achieve a very natural manner of building parsers, which combines the nicest properties of the functional and logic approaches. In particular, we are able to easily define within our framework parsers in a style very close to the 'monadic parsers' of functional programming, but using simpler concepts.

Keywords: parsing, functional-logic programming, non-deterministic functions.

1 Introduction

The problem of syntax analysis or *parsing* has been one of the most thoroughly studied issues in computer science (see e.g. [ASU86]). Its wide range of applications, from compilers development to natural languages recognition, is enough to attract the attention of any programming approach. This has been also the case for logic programming (LP, in short) and functional programming (FP, in short), and the parsing problem constitutes in fact one of the favorite fields for exhibiting the virtues of declarative programming.

Language processing has been very related to LP since its origins. There is a more or less standard approach [War80, CH87, SS86] to the construction of parsers in LP, which is based on a specific representation for grammars, the so-called *Definite Clause Grammars (DCG's)* [PW80, SS86]. *DCG*'s are not logic programs, although they are readily translated to them. With *DCG*'s, one can hide the details of handling the input string to be parsed, which is passed from parser to parser using the *Prolog* technique of *difference lists* [SS86]. Parsing in LP benefits from the expressive power of non-determinism and unification. We can mention:

• The ability to cope almost trivially with non-deterministic grammar specifications, as the multiple choice BNF construction '|'. The built-in backtracking mechanism will take charge of the search for a successful result. Also, multiple solutions are automatically provided where possible (e.g. ambiguous grammars).

• The skill in representing context sensitive languages.

Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid. e-mail: `rafacr@eucmos.sim.ucm.es`, `fraguas@eucmax.sim.ucm.es`

Work partially supported by the Spanish CICYT (project TIC 95-0433-C03-01 "CPD") and the ESPRIT Working Group 22457 (CCL-II).

• Simple construction of output representations, using pattern-matching and logical variables. The representation or translated code is carried out explicitly by using an input/output extra argument.

• Multiple modes of use: parsers may be used not only as recognizers but also as generators of sentences.

The most popular approach to writing parsers in FP is that of *parser combinators* [Wad85, Hut92, Fok95]. That is, parsers - starting from a set of basic ones – may be combined through carefully defined HO functions (the combinators) yielding new useful parsers. In the Haskell [HAS97] community, combinator parsing has derived recently into so-called *monadic parsing* [Wad90, Wad95, HM97].

Parsing with FP benefits of the power of types, functional notation and HO functions for writing clear, well-structured programs. As more concrete advantages we can mention:

• Parser combinators provide an incremental point of view of the compiler construction.

• HO combinators provide the ability of expressing BNF extensions, such as the repetitive application of the same parser zero or more times, which can not be expressed so cleanly in the LP setting.

• The use of *monads*, specially in combination with the *do notation* [Lau93, HM97] gives a very appealing structure to the parsers built up.

Many efforts have been done in the last decade in order to integrate LP and FP into a single paradigm, *functional-logic programming* (FLP in short, see [Han94] for a survey). As any other paradigm, FLP should develop its own programming techniques and methodologies, but little has been done from this point of view. We address here the problem of developing FLP parsers in a systematic way, trying to answer the question: can FLP contribute significantly by itself (not just mimicking LP or FP) to the task of writing parsers? This work should be then better understood as a contribution to FLP methodology, rather than from the perspective of compiler construction or parsing theory.

We stick to a particular view of FLP whose core notion is that of *non-deterministic function*. A framework for such approach was given in [GH+96], and later on extended to cope with higher-order features [GHR97], and polymorphic algebraic types in [AR97].

The rest of the paper is organized as follows. In the next section we will briefly describe the specific functional-logic language we are going to use: $\mathcal{TOY}$ [CLS97]. Section 3 presents a simplified format of our parsers – *parsers as recognizers* – whose only mission is to recognize valid sentences of the language defined by a grammar. These simple parsers serve to present the main ideas of our approach and make clear the differences with respect to the LP and FP cases. In Section 4 we discuss how to enhance parsers for obtaining, as a side product of recognizing a sentence, a suitable representation of it. We will arrive at this point to a style of writing parsers very close to the 'monadic' style and the 'do' notation of FP, but using much simpler constructs and without the need of any extra syntactic support. Finally, Section 5 summarizes some conclusions.

2 A succinct description of $\mathcal{TOY}$

All the programs in the next sections are written in the functional-logic language $\mathcal{TOY}$. We present here the subset of the language relevant to this work. A more complete description and a number of representative examples can be found in [CLS97].

A $\mathcal{TOY}$ program consists of *datatype*, *type alias* and *infix operator* definitions, and rules for defining *functions*. Syntax is mostly borrowed from Haskell [HAS97] (with the remarkable exception that variables begin with upper-case letters whereas constructor symbols use lower-case, as function symbols do). In particular, functions are *curried* and the usual conventions about associativity of application hold.

Datatype definitions like data nat = zero | suc nat, define new (possibly polymorphic) *constructed types* and determine a set of *data constructors* for each type. The set of all data constructor symbols will be noted as CS (CS^n for all constructors of arity n).

Types $\tau, \tau', \ldots$ can be constructed types, tuples $(\tau_1, \ldots, \tau_n)$, or functional types of the form $\tau \to \tau'$. As usual, $\to$ associates to the right. $\mathcal{TOY}$ provides predefined types such as [A] (the type of polymorphic lists, for which Prolog notation is used), bool (with constants true and false), int for integer numbers, or char (with constants 'a','b', ...). *Type alias* definitions like type parser_rec A = [A] $\to$ [A] are also allowed. Type alias are simply abbreviations, but they are useful for writing more readable, self-documenting programs. Strings (for which we have the definition type string = [char]) can also be written with double quotes. For instance, "sugar" is the same as ['s','u','g','a','r'].

The purpose of a $\mathcal{TOY}$ program is to define a set FS of functions. Each $f \in FS$ comes with a given *program arity* which expresses the number of arguments that must be given to f in order to make it reducible. We use FS^n for the set of function symbols with program arity n. Each $f \in FS^n$ has an associated principal type of the form $\tau_1 \to \ldots \to \tau_m \to \tau$ (where τ does not contain $\to$). Number m is called the *type arity* of f and well-typedness implies that $m \geq n$. As usual in functional programming, types are inferred and, optionally, can be declared in the program.

With the symbols in CS, FS, together with a set of variables $X, Y, \ldots$, we form more complex expressions. We distinguish two important syntactic domains: *expressions* and *patterns*. *Expressions* are of the form $e ::= X \mid c \mid f \mid (e_1, \ldots, e_n) \mid (e\ e')$, where $c \in CS$, $f \in FS$. As usual, application associates to the left and parentheses can be omitted accordingly. Therefore $e\ e_1 \ldots e_n$ is the same as $(\ldots((e\ e_1)\ e_2)\ldots)\, e_n)$. Of course expressions are assumed to be well-typed. *First order patterns* are a special kind of expressions which can be understood as denoting data values, i.e. values not subject to further evaluation, in contrast with expressions, which can be possibly reduced by means of the rules of the program. Patterns $t, s, \ldots$ are defined by $t ::= X \mid (t_1, \ldots, t_n) \mid c\, t_1 \ldots t_n$, where $c \in CS^n$.

Each function $f \in FS^n$ is defined by a set of conditional rules of the form

$$f\ t_1 \ldots t_n\ = e\ \Longleftarrow\ e_1 == e'_1, \ldots, e_k == e'_k$$

where $(t_1 \ldots t_n)$ form a tuple of linear (i.e., with no repeated variable) *patterns*, and e, e_i, e'_i are *expressions*. No other conditions (except well-typedness) are imposed to function definitions. Rules have a conditional reading: $f\ t_1 \ldots t_n$ can be reduced to e if all the conditions $e_1 == e'_1, \ldots, e_k == e'_k$ are satisfied. The condition part is omitted if $k = 0$.

The symbol $==$ stands for *strict equality*, which is the suitable notion (see e.g. [Han94]) for equality when non-strict functions are considered. With this notion a condition e == e' can be read as: e and e' can be reduced to the same pattern. When used in the condition of a rule, $==$ is better understood as a constraint (if it is not satisfiable, the computation fails), but the language contemplates also another use of $==$ as a function, returning the value true in the case described above, but false when a clash of constructors is detected while reducing both sides

As a syntactic facility, $\mathcal{TOY}$ allows repeating variables in the head of rules, but in this case repetitions are removed by introducing new variables and strict equations in the condition of the rule. As an example, the rule f X X = 0 would be transformed into f X Y = 0 $\Longleftarrow$ X == Y.

In addition to ==, $\mathcal{TOY}$ incorporates other predefined functions like the arithmetic functions +,*, ..., or the functions if_then and if_then_else, for which the more usual syntax if _ then _ and if _ then _ else _ is allowed. Symbols ==,+,* are all examples of *infix operators*. New operators can be defined in $\mathcal{TOY}$ by means of *infix* declarations, like infixr 50 ++ which introduces ++ (used for list concatenation, with standard definition) as a right associative operator with priority 50. Operators for data constructors must begin with ':', like in the declaration infix 40 :=. *Sections*, or partial applications of infix operators, like (==3) or (3==) are also allowed.

A distinguished feature of $\mathcal{TOY}$, heavily used along this paper, is that no confluence properties are required for the programs, and therefore functions can be *non-deterministic*, i.e. return several values for given (even ground) arguments. For example, the rules coin = 0 and coin = 1 constitute a valid definition for the 0-ary non-deterministic function coin. A possible reduction of coin would lead to the value 0, but there is another one giving the value 1. The system would perform first the first one, but if backtracking is required by a later failure or by request of the user, the second one would be tried. Another way of introducing non-determinism in the definition of a function is by putting *extra* variables in the right side of the rules, like in z_list = [0|L]. Any list of integers starting by 0 is a possible value of z_list. But note that in this case only one reduction is possible.

Our language adopts the so called *call-time choice* semantics for non-deterministic functions, following [Hus93, GH+96]. Call-time choice has the following intuitive meaning: given a function call $(f\ e_1 \ldots e_n)$, one chooses some fixed value for each of the e_i before applying the rules for f. As an example, if we consider the function double X = X+X, then the expression (double coin) can be reduced to 0 and 2, but not to 1. As it is shown in [GH+96], call-time choice is perfectly compatible with non-strict semantics and lazy evaluation, provided *sharing* is performed for all the occurrences of a variable in the right-hand side of a rule.

Computing in $\mathcal{TOY}$ means solving *goals*, which take the form $e_1 == e'_1, \ldots, e_k == e'_k$, giving as result a substitution for the variables in the goal making it true. Evaluation of expressions (required for solving the conditions) is done by a variant of lazy narrowing based on the so-called *demand driven strategy* (see [LLR93]). With respect to higher-order functions, a first order translation following [Gon93] is performed.

3 Parsers as recognizers

In this section we present a first simplified approach to our functional-logical parsers. The functions we are going to define represent syntactic analyzers, that is, they recognize whether a sentence belongs to a formal language or not, without performing any additional task. In later sections we will enhance their possibilities, retrieving representations whenever the sentence is successfully parsed.

Functional-logic parsers take as input parameter a list of terminal symbols, representing a possible sentence of the language. If some prefix of the list is successfully parsed, the rest of the list is returned as output value. Otherwise the parser fails, enforcing backtracking whether possible alternatives remain unused. Therefore, the type of our first parsers is:

```
type parser_rec A = [A] → [A]
```

that is, *parsers as recognizers* take a list of type A and return also a list of type A, A standing for *char* in most of the examples in this paper. To parse a sentence S through a parser P, we must try the goal P S == [ ], which succeeds if P parses successfully the whole sentence S.

In the following paragraphs the basic components of the functional-logic parsers as recognizers are defined, and a number of examples are included to illustrate their characteristics. The names of these components are partially borrowed from the literature on FP parsers, mainly those described in [Fok95].

Basic Parsers. We begin by defining the simplest parser: the empty parser, which just recognizes the empty word. This word is prefix of every sentence in any formal language, and therefore empty always succeeds, remaining the input list unaffected as output value.

```
empty :: parser_rec A
empty L = L
```

The next function takes a terminal symbol T and returns the parser that recognizes T if it is in the head of the input list, failing otherwise:

```
terminal:: A → parser_rec A
terminal T [T|L] = L
```

The output value is the rest of the list, discarding the recognized symbol T.

Sometimes it is desirable to recognize not a fixed symbol, but any one fulfilling a given property P. Function satisfy accomplishes this aim:

```
satisfy:: (A → bool) → parser_rec A
satisfy P [X|L] = if P X then L
```

For example, the parser digit = satisfy is_digit recognizes every string whose first character is in the set {'0',...,'9'}. Function terminal appears now as a specialization of satisfy, regarding the alternative definition: terminal X = satisfy (X==).

Parser Combinators. We have already defined the basic pieces we will use to build our parsers. What is needed now, in order to join the basic parsers in different ways, is a set of *parser combinators*. A parser combinator is a higher-order function which takes parsers as input parameters and returns a new parser as output value. The two main parser combinators we are going to define are the *alternative* and the *sequence* combinators.

The non-deterministic function '<|>', which we call the *alternative combinator*, captures the non-deterministic essence of BNF representations, and corresponds to the BNF construction '|'.

```
infixr 10  <|>
(<|>):: parser_rec A → parser_rec A → parser_rec A
(P1 <|>P2) L = P1 L
(P1 <|>P2) L = P2 L
```

The first choice of the infix operator <|> is its first parameter *P1*. Later, if the parsing process fails or more solutions are requested by the user, the parser *P2* will be selected using backtracking. Indeed, the operator <|> can be avoided using different rules for each alternative, i.e. instead of P = P1 <|>P2 we could write two rules for P, namely P = P1 and P = P2. At this point of the discussion both options may be regarded as equivalent, although we will see soon that they are not exactly the same.

The sequence combinator <∗> introduces the consecutive application of two parsers:

```
infixr 20 <∗>
(<∗> ):: parser_rec A → parser_rec A → parser_rec A
(P1 <∗> P2) L = P2 O1 ⟸ P1 L == O1
```

The use of the strict equality in the condition P1 L == O1 ensures that the application of the first parser P1 to the input L will be fully evaluated before applying the second parser P2 to the resulting output O1. The infix declarations previous to the definitions of <∗> and <|> fix their priorities as to minimize parentheses when combining them, and establish that both operators associate to the right. Thus we may combine easily the two combinators, defining for instance the parser

```
palin = empty  <|>  a  <|>  b  <|>  a <∗> palin <∗> a  <|>  b <∗> palin <∗> b
```

where a, b are defined as a = terminal 'a' and b = terminal 'b', respectively. This parser recognizes the language of the palindrome words over the alphabet $\Sigma = \{a, b\}$. Using this parser we may try the goal palin "abbababba" == [] which retrieves the answer yes meaning that the sentence belongs to the language, while the goal palin "abb" == [] fails because *abb* is not a palindrome.

Parsers as generators. Owing to the possibility of including logic variables in goals, FLP parsers may be regarded as generators as well as recognizers. For example, the goal palin [X,Y] == [] 'asks' for sentences of length two in the language recognized by *palin*. Two answers are retrieved, namely X='a', Y='a' and X='b', Y='b', meaning that *"aa"* and *"bb"* are the only words of length two in this language.

More higher-order parser combinators. To increase the expressiveness of the parsers, the extended BNF construction $\{P\}$ representing zero or more repetitions of the same parser P, may be defined. We call it the *star* of the parser P:

```
star:: parser_rec A → parser_rec A
star P = P <∗> (star P)  <|>  empty
```

For example we may define an identifier as a letter followed by a sequence of zero or more letters and digits: identifier = satisfy is_letter <∗> star (satisfy is_letter <|>satisfy is_digit) assuming a suitable definition of is_letter.

Despite the simple and natural definition of star, it contains one remarkable subtlety. Consider the following two rules defining the non-deterministic parser ab:

```
ab = terminal 'a'          ab = terminal 'b'
```

```
if_sent    =  terminals "if " <*> condition<*> terminals " then "
              <*> number <*> terminals " else " <*> number

condition  =  number <*> operator <*> number

operator   =  (terminal '<')  <|>  (terminal '>')  <|>
              (terminals ">=")  <|>  (terminals "<=")
```

Figure 1: Parser for simple *if* sentences.

that is, parser ab recognizes either one letter *a* or one letter *b*. Thus, the parser star ab should recognize any sequence of *a*'s and *b*'s. But if we try the simple goal (star ab) "ab" == [] the answer returned is no. This surprising behaviour is due to the two occurrences of the variable P in the body of star and it is related to the 'call-time choice' semantics adopted for non-deterministic functions. Specifically, if ab takes any value, either terminal 'a' or terminal 'b', 'call-time choice' fixes the same value for all the occurrences of ab in the computation of star ab. Hence, the parser star ab only can recognize sentences of the form *aaa...* or *bbb....*

The problem disappears if we use the operator <|> in the definition of ab:

```
ab = terminal 'a'  <|>  terminal 'b'
```

Now the computed value of ab is neither terminal 'a' nor terminal 'b' as before, but the irreducible expression terminal 'a' <|> terminal 'b'. The definition of <|> explains this fact: the operator needs an extra parameter (i.e. the input sentence) before selecting any of its arguments Thus, the non-deteministic choice is delayed and the repetition of the variable P in the body of star implies no more troubles. Consequently, whenever we want to use the combinator star, parsers must rely on the function <|> to handle non-determinism, instead of defining different rules for the same non-terminal.

Sometimes is useful to represent the repetition of a given parser one or more times instead of zero or more times. The combinator some accomplishes this aim:

```
some:: parser_rec A → parser_rec A
some P = P <*> (star P)
```

For instance, a number can be thought as a sequence of one or more digits: number = some digit where digit has been defined above.

Example: Parser for simple *if* sentences. Putting together what we have so far, we can define the parser shown in Figure 1. Conditional expressions are regarded as comparisons between two natural numbers using a relational operator, while the bodies of the parts *if* and *else* are natural numbers, with number the parser defined above. This figure introduces the new parser terminals, with type terminals:: [A] → parser_rec A , that generalizes terminal by recognizing a list of terminals instead of a single one. Its definition, terminals L = foldr (<*>) empty (map terminal L), relies on the standard functions foldr and map, and constitutes a typical example of how FLP inherits the higher-order machinery usual in FP. The goal if_sent "if 25>10 then 7 else 666" == [] succeeds, showing that the sentence is in the language recognized by if_sent.

Context sensitive languages. The previous examples show how FLP parsers can recognize context-free grammars following a notation close to BNF-rules. Now we are going to define a parser for the formal language $a^n b^n c^n$, showing that these parsers, by profiting from the virtues of logical variables, share with LP parsers the skill in recognizing context sensitive grammars. The abc parser may be seen in Figure 2.

```
data nat            =   zero | suc nat
abc                 =   count 'a' N <*> count 'b' N <*> count 'c' N
count _ zero        =   empty
count C (suc N)     =   terminal C <*> count C N
```

Figure 2: Parser for the context sensitive language $a^n b^n c^n$.

The parser count C N recognizes sequences of zero or more repetitions of the terminal C. It has an argument N of type nat expressing how many times the letter is repeated. The main parser abc employs a fresh variable N to enforce matching in the number of letters consumed by each parser. The role of each parser is clear here: N becomes instantiated when the parser count 'a' N acts, and then is used guiding the parsers count 'b' N and count 'c' N.

Comparison with FP and LP parsers. FLP parsers present several differences with FP and LP parsers, and here we sketch some of them. Although we have only presented the parsers as recognizers, the following discussion is valid also for the parsers presented in further sections.

We must point out the following differences between the FP and the FLP parsers:
• Observe that the type definition of FP parsers as recognizers would be *[A] → [[A]]*, that is, they would return *lists of results* containing together all the results that our parsers return in different computations using non-determinism. This difference leads to simpler and more natural definitions for FLP parsers.
• Parsing context sensitive languages is easier using FLP parsers than using FP parsers. For example, suppose we would like to define the parser presented in Figure 2 using FP. Then, the parser count C N could be used for parsing the sequences of b's and c's, but not for the a's, since the number of a's is not known in advance. Therefore different parsers should be used for a's, and for b's and c's.
• FLP parsers might be considered both recognizers and generators, while FP parsers are just recognizers. However, the possibility of generating sentences is widely used in LP parsers, for example when dealing with natural languages, as showed in [AD89].

In contrast, the main differences between FLP and LP parsers are:
• LP parsers as recognizers are modeled by means of predicates with two arguments, the first one for the input and the second one for the output. If we had adopted the LP point of view, parsers would have the type *[A] → [A] → bool*.
• Although some attempts have been developed (e.g. [Abr88]), introducing higher-order combinators in LP parsers is not easy (at least in *pure* LP). This entails the necessity of transforming the grammar rules into clauses using some meta-interpreter, as in the case of *Prolog* formalism of *DCG*'s. Conversely, FLP parsers are directly functions of the language and no extension of the language is needed. Moreover, FLP parsers allow

defining new combinators when necessary, while introducing, for instance, the combinator star using *DCG*'s is not directly allowed.

4 Parsers with representation.

Usually the parsing process is required to perform two different tasks, namely, checking whether the input string is a valid sentence of the formal language (that is what we have accomplished so far) and building a certain representation of the parsed string (e.g. the parsing tree). This section is devoted to the second point.

Thus, we need to associate some representation to parser functions. In FLP there are two alternative ways of returning values:

The FP solution, returning the representation as an output value. Notice, however, that our parsers return an output value yet and hence we need to combine the two values, the non-parsed part of the string and its representation, into a single output value. The natural solution is returning a pair of values. Therefore, given the type of the representation *Repr* and the type of the elements of the parsed list *Token* the parametrized parser type might be type parser Repr Token = [Token] $\rightarrow$ (Repr, [Token]) meaning that parsers will return a pair of values, whose first component is the representation of the parsed sentence, while the second one is the yet not parsed part of the sentence.

The LP solution. In LP all the values need to be parameters, and thus representations will be output parameters. In this case the type of parsers will be type parser Repr Token = Repr $\rightarrow$ [Token] $\rightarrow$ [Token] that is, a parser recognizes the sentence whose representation is given as a parameter. Actually, when recognizing sentences the parameter is just an unbounded variable which retrieves the representation of the parsed string.

Although the first type seems the natural choice, it needs some *plumbing* definitions when building new parsers. This problem has been overcome in FP using *monads* [Wad95], and adding some syntactic support in order to make the resulting expressions easier to read (e.g. the *do*-notation [Lau93, HM97]). We have adopted instead the second point of view, and we pretend to show here how this choice, carrying the representation as an output parameter, provides FLP parsers with most of the benefits that *monads* do with FP parsers. Moreover, no syntactic support is needed in our approach, nor even lambda abstractions are required.

Thus, the selected type for FLP parsers follows the FP approach for output lists, but the LP approach for representations:

type parser Repr Token = Repr $\rightarrow$ [Token] $\rightarrow$ [Token]

From now on, we call parses as recognizers just *recognizers*, in order to distinguish them from the parsers presented in this section which are named simply as *parsers*.

We are going to introduce representations in two stages: first we redefine the previous basic parsers and parser combinators, providing them with a *default representation*. Then we introduce a new combinator *do*, which allows parsers to include specific representations. This combinator will be used instead of the sequence combinator <$*$> whenever the default representation does not seem suitable.

Default Representations. Here we define a new set of basic parsers and parser combinators for dealing with values of type *parser*. They may be thought as upgrades of the functions we defined for recognizers in the previous section. For this reason the same function names are used, which should not be confusing as from now on only the new definitions are considered.

The new basic parsers and parser combinators have the same meaning that their previous versions, but they also include and handle *default representations*. These representations will be built automatically, providing new parsers with 'representations for free', which may be valuable whenever we agree with the values provided or we do not care about representations. But, which should be the default representation when parsing a sentence? The default representation of a sentence is ... the sentence itself. This feature may be checked easily examining the upgraded set of basic parsers and parser combinators showed in Figure 3. In fact, it is not difficult to prove inductively the property $p\,R\,I == O \implies R ++ O == I$, for any parser p built up from functions in the figure, and for any R, I, O lists of terminals finite and totally defined.

```
empty:: parser [A] A
empty [] L           =   L

satisfy:: (A → bool) → parser [A] A
satisfy C [X] [X|R]  =   if C X then R

terminal:: A → parser [A] A
terminal [T] [T|R]   =   R

(<*>):: parser [A] B → parser [A] B → parser [A] B
(P1 <*> P2) R L      =   P2 R2 O1 ⇐= P1 R1 L == O1, R1++R2 == R

(<|>):: parser A B → parser A B → parser A B
(P1 <|> P2) R L      =   P1 R L
(P1 <|> P2) R L      =   P2 R L
```

Figure 3: Basic parsers and parser combinators with representation.

To become parsers, the rest of the recognizers we have defined so far need only to change their types from *parser_rec* to *parser* including the type of the new representations. The function rules do not need any change as a nice outcome of the technique of default representations. For example the definition of function *star* is still valid, and we only need to change its type to star:: parser [A] B → parser [A] B. The <∗> , empty and <|> functions provide function *star* with its own default representation, which is a list whose elements are the representations retrieved for each occurrence of P. Consider for instance the recognizer defined in Figure 1. With the new definitions, it may be regarded as a function of type *parser*, and simply try the goal if_sent R "if 25>10 then 7 else 666" == [] which returns yes with R=="if 25>10 then 7 else 666".

It can be argued that such a representation, the same sentence that we have parsed, is useless, but in support of this technique we must point out that:

1. It provides an incremental point of view of the compiler development. The default representation may be used in a first stage of the development, when we are inter-

ested in the language itself, ignoring representations. Later, suitable representations can be included with only a few changes in the code.

2. When defining the final representation, some of the default constructions are surely going to be kept unaffected. For example, is not likely that we prefer other representation for an identifier rather than its own name. This mix of *user provided* and *default* representations may quicken the development process.

Providing general representations. Now we are going to set up the new constructions *do_s* and *do*. They allow parsers to include specific representations, replacing the sequence combinator <∗> whenever we are not interested in the default representation.

The first construction introduced is the *simple do*, represented as *do_s*. The next example shows the purpose and usefulness of this construction and will help when understanding its definition. The parser with default representation *binding* recognizes variable assignments of the form var1 = var2; : binding = identifier <∗> terminal '=' <∗> identifier <∗> terminal ';' where identifier and terminal are now regarded as of type *parser*. Suppose we decide that a suitable representation for an assignment may be a value of the data type data bind_rep = [char] := [char], where := is an infix constructor and the two lists standing for the names of the variables. Then we may define

```
binding (V1:=V2) = do_s [identifier V1, terminal '=' _, identifier V2, terminal ';' _]
```

We have included in the same list all the parsers that were connected by <∗> , providing each parser with an argument standing for its representation. The final aspect of the function is not very different from that of LP parsers: the representation of the parser appears in the shape of a pattern, (*V1:=V2*), whose variables are the representations of its component parsers. The dummy variables (_) mean that we are not going to use those representations. Note that we still rely on the default representations for *identifier*, as this value is the very representation we need, i.e. the name of the variable.

As we said before *identifier* is of type *parser*. Therefore *identifier V1* is of type *parser_rec*, and the same is valid for all the functions in the list. Thus, the combinator *do_s* may be seen as a generalization of the sequence combinator for recognizers, though here it is used in a quite different context. The next definition should be now understable:

```
do_s::[parser_rec A] → parser_rec A
do_s [] Input       =  Input
do_s [X|Xs] Input   =  do_s Xs O1 ⟸ X Input == O1
```

Function *do_s* eases the construction of certain representations, but it enforces representations to be *patterns*. What is needed to build general representations is to combine all the intermediate values through a fixed expression *Exp*.

Such generalization of *do_s* is provided by the construction *do*. It takes a list *L* of recognizers and an expression *Exp* as input values, and returns the parser that recognizes the same sentences that the elements in the list when connected in sequence, and whose representation is the result of evaluating *Exp* after recognizing the sentence.

```
do::[parser_rec A] → B → parser B A
do L Exp Rep Input = O ⟸ do_s L Input == O, Exp==Rep
```

```
if_sent     =   do [terminals "if " _, condition C, terminals " then " _,
                num N1, terminals " else " _, num N2] (if C then N1 else N2)

condition   =   do [num N1, operator Op, num N2] (Op N1 N2)

operator    =   (terminal '<')⟶(<)  <|>  (terminals "<=")⟶(<=)  <|>
                (terminal '>')⟶(>)  <|>  (terminals ">=")⟶(>=)
```

Figure 4: Parser for simple *if* sentences with representation.

For instance, the best representation of a number is surely its numeric value, as is settled in parser num: num = do [some digit L] (num_value L) . The representation of num is therefore the result of applying the function num_value to the default representation L of some digit, where digit is the parser defined before. The function may be defined as num_value L = foldl ((+)·(10×)) 0 (map val L), where val is a function that converts a single digit in its numeric value, and foldr, *the composition (.)* and map, are standard functions.

Example: simple *if* sentences with representation. The final version of the parser for the *if* sentences is presented in figure 4. The representation now is the result of evaluating the sentence, that is, the number in the *if* part if the condition is satisfied, or the number in the *else* part otherwise.

Note the inclusion of the new construction ⟶ in the definition of operator. This construction may be useful when the final representation of a parser does not depend upon intermediate values. It takes as input parameter one parser P and its desired representation R. Then applies the parser, and returns R as output representation, dismissing the representation retrieved by P. It may be defined in the following, straightforward way

```
infixr 30 ⟶
(⟶):: parser A B → C → parser C B
(P ⟶ R) R = P _
```

In this case each operator uses ⟶ to return as representation the operator itself, regarded as a partial function. The representation of the parser condition is the result of applying the operator to the two numbers returned by the previously defined parser num. Finally the representation of if_sent is determined by the expression if C then N1 else N2. The goal if_sent R "if 25>10 then 7 else 666" == [] returns now R == 7.

Observe that our construction do is very similar to that of FP parsers. However, ours is simply a HO function, whereas the *do* of FP needs a specific syntactic support.

It has been argued [Pre96] that the non-determinism of a grammar can be turned out into determinism by means of adding an extra argument to the deterministic FLP parsers. This extra argument is something similar to a parse tree that distinguishes the different alternatives chosen while parsing the input string. We claim that this solution is less effective than ours, as it limits the possible representations retrieved by parsers, and builds large structures of nested alternatives with unnecessary information.

5 Conclusions.

We have shown how a functional-logic language supporting non-deterministic functions allows defining parsers which combine most of the nicest properties of both functional and logic parsers. Specifically, FLP parsers share with LP parsers the natural way of handling non-determinism provided by non-deterministic computations, the skill in recognizing context sensitive languages, and the possibility of multiple modes of use. On the other hand, FLP parsers profit from many FP features, as the definition of powerful HO combinators or the use of functional types. For the problem of constructing involved representations of the parsed sentences, we have proposed a technique (our `do` construction) resembling FP monads in the style of parsers that can be written, but with the advantage of not needing any extra syntactic support. Actually, this similarity may deserve a thorough study, for it suggests that our technique could be generalized to other areas where monads have been employed successfully. For the sake of space, we have not discussed here other issues related to parsing which can be addressed successfully in our FLP framework. A remarkable one is the possibility, by making use of *higher order patterns* in rules, of managing *parsers as data*, in such a way that interesting properties – for instance, if the underlying grammar is $LL(1)$ – can be examined (see [CL98] for details).

Acknowledgements:
We thank Mario Rodríguez-Artalejo for many valuable comments about this work.

References

[Abr88] H. Abramson. *Metarules and an Approach to Conjunction in Definite Clause Translation Grammars: Some Aspects of Grammatical Metraprogramming* . Logic Programming, Procs. of the Fifth Interantional Conference and Symposium . 1988. R.A. Kowalski and K.A. Bowen (eds.), pp 233-248.

[AD89] H. Abramson, V. Dahl, *Logic Grammars.* Springer-Verlag, 1989

[AR97] P. Arenas-Sánchez, M. Rodríguez-Artalejo. *A Semantic Framework for Functional Logic Programming with Algebraic Polymorphic Types.* Procs. of CAAP'97, Springer LNCS 1214, 453–464, 1997.

[ASU86] A. V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addisson-Wesley, 1986.

[CH87] J. Cohen, T. Hickey. *Parsing and Compiling using Prolog*, ACM TOPLAS 9 (2), 1987, 125–163.

[CLS97] R. Caballero-Roldán, F.J. López Fraguas and J. Sánchez-Hernández. *User's Manual For $\mathcal{TOY}$* . Technical Report D.I.A. 57/97, Univ. Complutense de Madrid 1997. The system is available at `http://mozart.sip.ucm.es/incoming/toy.html`

[CL98] R. Caballero-Roldán and F.J. López Fraguas. *Functional-Logic Parsers in $\mathcal{TOY}$* . Technical Report S.I.P. 74/98. Univ. Complutense de Madrid 1998. Available at `http://mozart.sip.ucm.es/papers/1998/trparser.ps.gz`

[Fok95] J. Fokker. *Functional Parsers.* In J. Jeuring and E. Meijer editors, Lecture Notes on Advanced Functional Programming Techniques, Springer LNCS 925, 1995.

[GH+96] J.C. González-Moreno, T. Hortalá-González, F.J. López-Fraguas, M. Rodríguez-Artalejo. *A Rewriting Logic for Declarative Programming.* Procs. of ESOP'96, Springer LNCS 1058, 156–172, 1996.

[GHR97] J.C. González-Moreno, T. Hortalá-González, M. Rodríguez-Artalejo. *A Higher Order Rewriting Logic for Functional Logic Programming.* Procs. of ICLP'97, The MIT Press, 153–167, 1997.

[Gon93] J.C. González-Moreno. *A Correctness Proof for Warren's HO into FO Translation.* Procs. of GULP'93, 569–585, 1993.

[Han94] M. Hanus. *The Integration of Functions into Logic Programming: A Survey.* J. of Logic Programming 19-20. Special issue "*Ten Years of Logic Programming*", 583–628, 1994.

[HAS97] *Report on the Programming Language Haskell: a Non-strict, Purely Functional Language.* Version 1.4, Peterson J. and Hammond K. (eds.), January 1997.

[Hus93] H. Hussmann. *Non-determinism in Algebraic Specifications and Algebraic Programs.* Birkhäuser, 1993.

[Hut92] G. Hutton. *Higher-Order Functions for Parsing.* J. of Functional Programming 2(3):323-343, July 1992.

[HM97] G. Hutton, E. Meijer. *Functional Pearls. Monadic Parsing in Haskell.* To appear in J. of Functional Programming. Extended version: Tech-Rep NOTTCS-TR-96-4. Dept. of Computer Science. Univ. Nottingham ,1996

[Lau93] J. Launchbury. *Lazy imperative programming.* In Procs. ACM Sigplan Workshop on State in Programming Languages, YALE/DCS/RR-968, Yale University, 1993.

[LLR93] R. Loogen, F.J. López-Fraguas,M. Rodríguez-Artalejo. *A Demand Driven Computation Strategy for Lazy Narrowing.* Procs. of PLILP'93, Springer LNCS 714, 184–200, 1993.

[Pre96] C. Prehofer. *Solving Higher-Order Equations. From Logic to Programming. Progress in Theoretical Computer Science*, Birhäuser, 1998.

[PW80] F. Pereira, D.H.D. Warren. *Definite Clause Grammars for Language Analysis*, Artificial Intelligence 13, 1980, 231–278.

[SS86] L. Sterling, E. Shapiro. *The Art of Prolog*, The MIT Press, 1986.

[Wad85] P. Wadler. *How to Replace Failure by a List of Successes*, Proc. IFIP FPCA'85, Springer LNCS 201, 1985, 113–128.

[Wad90] P. Wadler. *Comprehending Monads*, Proc. ACM Conf. on Lisp and Functional Programming, 1990.

[Wad95] P. Wadler. *Monads for functional programming.* In J. Jeuring and E. Meijer editors, Lecture Notes on Advanced Functional Programming Techniques, Springer LNCS 925. 1995

[War80] D.H.D Warren. *Logic Programming and Compiler Writing*, Software Practice and Experience 10, 1980, 97–125.