FUNCTIONAL-LOGIC PARSERS IN $\mathcal{TOY}$[*]

Rafael Caballero Roldán - Francisco J. López Fraguas

TECHNICAL REPORT SIP - 7498

Dep. Lenguajes, Sistemas Informáticos y Programación
Univ. Complutense de Madrid

APRIL 98

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 modelling 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 without any syntactic overhead. In a different direction we show that, if the functional-logic setting permits higher-order patterns while defining functions, parsers can be freely manipulated as data. This allows programming useful metalevel analysis or transformations, like discovering if the grammar corresponding to a given parser is $LL(1)$. Finally, we sketch the potential application to parsing of functional-logic programming extended with numerical constraints.

[*]The authors have been partially supported by the Spanish CICYT (project TIC 95-0433-C03-01 "CPD") and the ESPRIT Working Group 22457 (CCL-II).

Contents

1 Introduction.

1.1 The parsing problem.

The problem of the 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 compiler development to natural languages recognition, is enough to attract the attention of any programming approach. Declarative programming has also tackled this problem, mainly looking for a straightforward way of representing parsers as proper components of the language. This has been reached considering *recursive descendent parsers* usually represented by means of language mechanisms adapted to simulate grammar rules (e.g. BNF rules). Actually we could say that declarative languages have yield new perspectives to the parsing theory, but it seems also clear that parsing has supplied new features to declarative languages. Indeed, *Prolog* *D*CGs [SS86, PW80] are part of the language only for parsing purposes, while one of the first applications of the functional programming *monads*[Wad85, Wad90] were the parser combinators [Wad95].

However the two most important declarative programming paradigms, namely functional and logic programming (in short FP and LP), have considered different approaches to the parsing problem. We will first survey the main characteristics of parsing in each paradigm.

1.2 The Logic Programming approach.

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].

LP Parsers benefit from the expressive power due to non-determinism and logical variables. Briefly, the main characteristics of these parsers are:

1. The ability to cope easily 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.

2. Multiple solutions are automatically provided where possible (e.g. ambiguous grammars).

3. The possibility of representing context-sensitive languages.

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

5. Parsers may be not only recognizers for formal languages but also generators of sentences.

1.3 The Functional Programming approach.

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:

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

2. 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.

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

1.4 The Functional-Logic Programming approach.

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 affirmatively to the question: can FLP contribute significantly by itself (not just mimicking LP or FP) to the task of writing parsers?

We stick in our work 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]. In the rest of the report we show the characteristics of functional-logic parsers, (FLP parsers), which amalgamate most of the benefits of the two paradigms showed above. In addition, the use of constraints and higher-order patterns will provide new possiblities to the parsing process. The main characteristics of the FLP parsers we present are:

1. Non-deterministic grammar definitions are easily handled by means of non-deterministic functions.

2. Some context sensitive languages can be represented.

3. Representations can be constructed either by pattern matching or through defined functions.

4. The parsers can be used either as recognizers or as generators of sentences.

5. Higher order combinators may be defined, in a way close to the functional approach.

6. Numerical constraints allow parsing some special languages (e.g. visual languages).

7. Higher order patterns allow to program the verification of some properties of the defined grammars: set of *first* or *follow* symbols, $LL(1)$ condition, etc. Thus funcional-logic parsers may be also been considered as manageable *data*.

The next section is devoted to describe briefly an specific functional-logic language: $\mathcal{TOY}$. A first attempt of building functional-logic parsers is then presented, the *generate and test parsers*. Although naive and not powerful enough yet, these parsers will include some of the main characteristics of the final *input-output parsers*, which are introduced in section 4. Section 5 is devoted to the presentation of an example, showing the application of numerical constraints to some special kind of languages. Finally, in section 6 we will meet the use of higher-order patterns to check interesting characteristics of our grammars, such as the *first* and *follow* sets of symbols. Several examples written in $\mathcal{TOY}$ are included to illustrate the points discussed throughout the report.

2 The base language.

All the examples we show in the rest of the report are written in the functional-logic language $\mathcal{TOY}$[1]. 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].

$\mathcal{TOY}$ is a non-deterministic functional-logic language developed at the Universidad Complutense de Madrid. Functions in $\mathcal{TOY}$ are defined by *conditional equations* and are executed by *lazy narrowing* and *backtracking.* 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*

[1]The system is available at `http://mozart.sip.ucm.es/incoming/toy.html`

A = *[A]* $\rightarrow$ *[A]* are also allowed. Type alias are simply abbreviations, but they are useful for writing more abstract, 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 $\tau_1 \rightarrow \ldots \rightarrow \tau_m \rightarrow \tau$ (where τ does not contain $\rightarrow$). 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: *patterns* and *expressions*. Patterns 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 $t ::= X \mid (t_1, \ldots, t_n) \mid c\ t_1 \ldots t_n \mid f\ t_1 \ldots t_n$, where $c \in CS^m$, $n \leq m$, $f \in FS^m$, $n < m$. Notice that partial applications (i.e., application to less arguments than indicated by the arity) of c and f are allowed as patterns, which are then called *HO patterns*, due to their functional type. Therefore function symbols, when partially applied, behave as data constructors. HO patterns can be manipulated as any other patterns; in particular, they can be used for matching or checked for equality. *Expressions* are of the form $e ::= X \mid c \mid f \mid (e_1, \ldots, e_n) \mid e\ e_1 \ldots e_n$, where $c \in CS$, $f \in FS$. Of course expressions are assumed to be well-typed.

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 $k = 0$. For intance, the infix operator ++ takes two lists and appends the elements of the second list at the end of the first list:

```
infixr 50 ++
(++ ) [] Y        =   Y
(++ ) [X|Xs] Y    =   [X|Xs ++ Y]
```

The symbol == stands for *strict equality*, which is the suitable notion (see e.g. [Han94]) for equality when non-strict partial 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 the 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 [2]. Both uses of == are distinguishable by the context.

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 new strict equations in the condition of the rule. As an example, the rule

```
f X X = 0
```

would be transformed into

$$\mathsf{f\ X\ Y = 0 \Longleftarrow X == Y}$$

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 :=*.

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
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. An interesting example of non-deterministic function is the *choice operator.* This function takes two parameters and choices one of them as output result. It may be defined as follows:

```
infixr 10  //
(//) E1 E2      =   E1
(//) E1 E2      =   E2
```

The *infixr* declaration fix the priority of the function as infix operator (10) and its associativity (right).

[2]A full implementation of the function == requires to incorporate *disequality constraints*, like in [AGL94], and $\mathcal{TOY}$ indeed features them, but we will not use this aspect of the language.

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 a sophisticated strategy, called *demand driven strategy* in [LLR93], which uses the so-called *definitional trees* [Ant92] to guide unification with patterns in left-hand sides of rules (see [LLR93]). With respect to higher-order functions, a first order translation following [Gon93] is performed.

3 Generate & Test parsers.

3.1 Generate & Test recognisers

BNF-descriptions of grammars have most of the times some degree of non-determinism, because of the presence of different alternatives for the same non-terminal symbol. In logic programming this situation is easily handled by using non-deterministic computations. In a similar way we use *non-deterministic functions*, which will provide multiple alternatives by means of backtracking. Our formulation of BNF rules is even more natural than in the case of logic programs: no extra arguments nor preprocessing is needed.

Our first example, a simple parser written in $\mathcal{TOY}$ representing the BNF-rules for the language of the arithmetic expressions over 0 and 1, may be seen at figure 1, while the corresponding BNF rules are showed in the figure 2.

```
expression   =  term     //   term ++ plus_minus ++ expression
term         =  factor   //   factor ++ prod_divi ++ term
factor       =  const    //   "(" ++ expression ++ ")"
const        =  "0"      //   "1"
plus_minus   =  "+"      //   "-"
prod_divi    =  "*"      //   "/"
```

Figure 1: g&t parser for simple arithmetic expressions

Some points of this example deserve a detailed explanation:

- Parsers are defined by non-deterministic functions whose output values are all the sentences of the formal language.

```
<expression>   ::=  <term>      |  <term><plus_minus><expression>
<term>         ::=  <factor>    |  <factor><prod_div><term>
<factor>       ::=  <const>     |  ( <expression> )
<const>        ::=  0           |  1
<plus_minus>   ::=  +           |  -
<prod_div>     ::=  *           |  /
```

Figure 2: BNF rules for simple arithmetic expressions

- The alternatives are represented through the non-deterministic function // . By means of // we can represent the BNF construction | which denotates alternatives. Observe that we could have defined the non-deterministic functions directly as well, writting for instance:

  ```
  const  =  "0"
  const  =  "1"
  ```

  instead of:

  ```
  const = "0"  //  "1"
  ```

  However, we prefer to concentrate all the sources of non-determinism in the same function, namely the operator //. At this point of the discussion, this decision may be thought just as a question of programming style, but after defining the *parser combinators* other important reasons will become apparent.

- To combine two parsers in sequence it is only neccessary to append the two strings with the infix operator ++, defined elsewhere as the concatenation of two lists. Both *parser combinators*, ++ and //, can be combined in the same function as desiderable .

- In this example the *terminals* of the grammar are represented as concrete strings (eg. "*") and can be combined with the nonterminals using ++ and //.

For instance the parser $factor$ may be read as: "a factor is either a constant (*const*) or an open bracket followed by an expression and ended by a closed bracket". Hence we can say that the non-deterministic function *expression* is a functional-logic parser for the language.

To recognize that a concrete list of tokens L is a valid sentence of the language corresponding to the non-terminal symbol s, we simply try to generate L using the function s. This is accomplished by solving the goal

```
s == L
```

Backtracking allows the attempt of the different possibilities, until the desired sentence is generated and therefore the proccess of recognizing success, or all the alternatives are exhausted and the parser fails, meaning that the sentence is refused as part of the language. For obvious reasons we call this kind of parsers *generate & test* or in short, g&t parsers: instead of taking the string we would like to parse as an input parameter like usual, these parsers generate a sentence of the language, which is compared with the desired string.

An example of sentence successfully recognised using the g&t parser *expression* may be:

```
expression == "1*0+1+1"
```

The system yields the answer *yes* , meaning that the sentence "$1*0+1+1$" is in the language recognised by the parser *expression*, while in the goal

```
expression == "1**0"
```

the system retrieves *no*, because "1**0" is not a valid sentence of the language.

3.2 Context sensitive languages.

We have already shown 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 3.

```
data nat      =   zero | suc nat
abc           =   as N ++ bs N ++ cs N
as zero       =   ""
as (suc N)    =   "a" ++ as N
bs zero       =   ""
bs (suc N)    =   "b" ++ bs N
cs zero       =   ""
cs (suc N)    =   "c" ++ cs N
```

Figure 3: Parser for the context sensitive language $a^n b^n c^n$.

The parsers *as*, *bs* and *cs* represent sequences of zero or more terminals *'a'*, *'b'* and *'c'* respectively. These three parsers have 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 *as* acts, and then is used guiding the parsers *bs* and *cs.*

3.3 Generating sentences.

Another interesting feature shared between LP and FLP parsers is the skill in generating sentences of the formal language represented, instead of just recognizing them. In the next goal we ask for the valid sentences of the form $[X,'*',X]$, with X any valid terminal symbol.

```
expression == [X,'*',X]
yes
X == '0'

more solutions [y]?
yes
X == '1'

more solutions [y]?
no.
```

The answers provide the values of the logical variable X which satisfy the goal, namely 0 and 1. Thus the g&t parsers might be regarded as generators, recognizers or a mix of both possibilities, depending on the goal. Owing to this reason, from now on we indistinctly refer to the sentence *generated* or *recognised* by a given g&t parser.

As an aside we note that both the skill in generating sentences and the possibility of representing easily context sensitive languages, are not specific of the g&t parsers. Indeed, these features are shared by all the FLP parsers presented in this report, and the examples can be easily adapted to each notation we will use.

3.4 About performance.

In a first sight the g&t process seems rather awkward. As interesting languages are usually infinite, it doesn't appear possible determining whether the desired string will be produced in a finite lapse of time or not. Fortunately the situation is very different. The operational semantics of the strict equality assures that the output sentence will be compared with the desired string while it is produced, enforcing backtracking when coincidence is not possible. Actually, we could say that the string we would like to parse leads the output values of the parser, via strict equality. If these naive parsers are not powerful enough is due to the lack of expressiveness of the generated representation (see subsection [3.8]), not because of the inefficiency of the generate and test process.

Consider again the successful goal

```
expression == "1*0+1+1"
```

We are going to examine the first steps of the recognizing process:

1. Initially, the parser *expression* has two alternatives, represented by the choice function //. This function first choices its first argument, so the parser *term* is tried first. The parser *term* has also two possibilities, and for the same reason the first one, the parser $factor$ is choiced. Similarly $factor$ leads to $const$, and $const$ leads to 0.

2. Now we have a first output value, 0 which is compared with the first character of the given string, 1. The comparation fails, and the last choice, 0 for $const$, is rejected. The function // returns now, by backtracking its second choice, 1, and the comparation succeeds.

3. The next step is to compare the rest of the generated string with "*0+1+1". But 1 is a valid sentence and it has no more characters. Therefore, the rest of the generated sentence is the empty list which does not match "*0+1+1", enforcing backtracking.

4. The parser *const* has not more choices through //, and fails. The second possibility for *factor* is then tried.

5. Observe that although the current choice, "(" $++$ *expression* $++$ ")" could generate infinite sentences, *all of them are rejected* at once. Effectively, as soon as the sequence operator ++ returns the first character of the generated sentence, the open bracket, it is compared with the first character of the input string, 1, and the comparation fails.

6. The parser *factor* has not more choices and fails itself. Now the second choice of *term* is tried. The sequence operator ++ tries to get the first character from *factor*. It returns 0 in a first attempt as we saw previously, but after the failed comparation returns 1 which matches with the first input character.

7. Now there are two alternatives: the character $*$ or the character /. The function // choices the first one, $*$ which is successfully compared with the second character of the string we want to parse.

After a few steps more, the remaining input string, $0+1+1$, has been completely checked and the answer *yes* is displayed by the system.

As we have seen, the output sentences of the *expression* parser are compared with the string we would like to parse at the same time the values are generated. Consequently, unsuccessful (and some times infinite) branches of the search tree are pruned as soon as the first difference with the input string is realised.

Notice, however, that ocassionally during the parsing process a complete sentence that does not match with the desired string might be generated, as in the step 2. This situation, which affects the efficiency of the parser, occurs usually in grammars with productions of the form:

$$A \to \alpha\beta_1 \mid \alpha\beta_2$$

with α, β_1, β_2 sequences of terminals and non-terminals, and can be eluded by introducing a new non-terminal A' and left factoring the grammar:

$$\begin{array}{lcl} A & \to & \alpha A' \\ A' & \to & \beta_1 \mid \beta_2 \end{array}$$

Hence, we might introduce new nonterminals in order to left factorize our parser, but our notation allows us to go even further, avoiding the necessity of new nonterminals, as showed in figure 4.

3.5 Representations.

Usually the parsing process is required to perform two different tasks:

```
expression  =  term    ++  ( ""  //  plus_minus ++ expression)
term        =  factor  ++  ( ""  //  prod_div ++ term )
factor      =  const   //  "(" ++ expression ++ ")"
const       =  "0"     //  "1"
plus_minus  =  "+"     //  "-"
prod_div    =  "*"     //  "/"
```

Figure 4: left-factoring a g&t parser

1. Checking whether the input string is a valid sentence of the formal language or not. This is what we have accomplished so far.

2. Retrieving a certain representation of the parsed string (e.g. the parsing tree).

The first step is related to the *syntax analysis phase* of a compiler, while the second is close to the *code generation phase*. As the generated code or *representation* of a parser often depends upon the representations of its component parsers, it seems logical to carry out syntax analysis and code generation simultaneously [3]. Thus we need to associate some representations to the parser functions. In functional-logic programming there are two alternative ways of returning values:

1. As an output value. This seems the natural approach, and is the used in functional parsers. Notice, however, that our parsers are returning an output value currently, the generated string. Hence we need to combine the two output values, the parsed 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 = (Repr, [Token])
   ```

   meaning that a general parser will return a pair of values, whose first component is the representation of the parser sentence while the second is the sentence itself. It is worth noting that we allow any type as representation, but we enforce the generated/recognised sentence to be always a list of tokens (e.g. a list of chars as in the figure 1).

2. As an output parameter. This is the choice of logic programming. In this case the type of a general parser will be:

   ```
   type parser Repr Token = Repr ⟶ [Token]
   ```

   that is, a parser generates/recognises the sentence of the formal language whose representation is given as a parameter. Actually, when recognising sentences the parameter is just a unbounded variable which returns the representation of the parsed string.

[3]The basic idea we use is near to *syntax-directed* translation, as described in [ASU86]

Although the first type seems the natural choice, it leads to multiple problems when combining parsers. In fact, building explicit representations using the representations of the intermediate components requires the definition of some programming mechanism, in order to extract the representation from each intermediate output value. This problem has been overcome in functional programming 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]).

In this report we pretend to show how the second choice, carrying the representation as a function parameter, provides functional-logic parsers with the same benefits as monads do with functional parsers. Moreover, no syntactic support is needed in our approach, nor even lambda abstractions are required. Thus, the selected type for general parsers is the second one:

```
type parser Repr Token = Repr ⟶ [Token]
```

That is, the generate and test parsers will have a single parameter, the representation of the parsed string. The output value will be the generated sentence of the language corresponding to the given representation. If several sentences of the language have the same representation, or the representation itself is not a ground term and allows several instances, then the parser will provide all the possible answers using non-determinism.

Suppose we would like to supply some output representation to the parser for simple arithmetic expressions. For instance we could return the *precedence tree* of the expression by means of an appropiate data definition, like:

```
data operators   =   plus | minus | mult | divi
data prec_tree   =   val int   |   op operators prec_tree prec_tree
```

The data type *prec_tree* has two data constructors, *val* for representing the numeric value of a constant (i.e. 0 or 1), and *op* for representing expressions with infix operators. The first argument of *op* is a value of type *operators* which represents the main operator OP of the expression, while the second and the third arguments represent the subexpressions that are being operated through OP. The parser with representation may be seen in figure 5.

Note that it has been necessary to unfold some of the rules to distinguish the different representations that were encoded in the same function. Sometimes (e.g. in the parser *expression*) we do not need to write explicitly the output representation, as it is provided directly from the component parsers. In other cases (e.g. *add_expr*) the generated representation is built using the intermediate representations, which may be thought as *synthetized attributes.* For example the parser *factor* now may be read as: 'a *factor* with representation R is either a *const* with representation R or an *expression* with representation R and enclosed by parentheses'.

Using this parsers we may for instance recognize again the string "1*0+1+1", but now getting also the output representation of the parsed string.

```
expression R == "1*0+1+1"

R == op plus (op mult (val 1) (val 0)) (op plus (val 1) (val 1)))
yes
```

```
expression,add_expr:: parser prec_tree char
expression                  =   term  //  add_expr
add_expr (op Op T1 T2)      =   (term T1) ++ (plus_minus Op) ++ (expression T2)

term,mult_expr:: parser prec_tree char
term                        =   factor  //  mult_expr
mult_expr (op Op T1 T2)     =   (factor T1) ++ (prod_divi Op) ++ (term T2)

factor, const,zero,one:: parser prec_tree char
factor R                    =   (const R)  //  "(" ++ (expression R) ++ ")"
const                       =   zero  //  one
zero (val 0)                =   "0"
one (val 1)                 =   "1"

plus_minus, proc_divi :: parser operators char
plus_minus                  =   add_op  //  minus_op
prod_divi                   =   mult_op  //  div_op

add_op, minus_op, mult_op, div_op:: parser operators char
add_op plus                 =   "+"
minus_op minus              =   "-"
mult_op mult                =   "*"
div_op divi                 =   "/"
```

Figure 5: g&t parser with representation for simple arithmetic expressions

The answer *yes* denotes as usual that the string have been recognized as a valid sentence of the formal language, while the value of R retrieves the parsing structure of the sentence.

In the next subsections we present some basic parsers and parser combinators. Later we will use these basic pieces to build more complicated and useful parsers. These parsers will be the g&t versions of the functional parsers defined in [Fok95, Hut92].

3.6 Basic parsers.

We start by defining the *empty parser*, which just generates the empty sentence, being also the empty list its representation. The usefulness of this parser becomes apparent when combined with the choice combinator to define optional possibilities.

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

The next function takes an input value T and retrieves the parser that generates the list whose only element is T.

```
terminal::A → parser A A
terminal T T = [T]
```

The representation of the only sentence generated by the parser *terminal T* is the element T itself. For instance we may try the simple goal:

```
terminal 0 R == "0"
```

which parses successfully the string "0", retrieving the representation 0 in the variable R.

It may be also desiderable to define a non-deterministic function for generating all the sentences of length 1 satisfying a given property. We call this the *satisfy parser* and may be defined as follows:

```
satisfy::(A→ bool) → parser A A
satisfy P A = if P A then [A]
```

The parameter P represents a property over the elements of type A. The representation, the parameter A, is also the only element of the generated sentence. For instance, the goal:

```
satisfy is_digit R == "5"
```

yields the answer *yes*, with R instantiated to 5. Notice the possible non-determinism of the parser *satisfy P* for each given parser P: it may generate as many sentences as different values satisfy the function P. For instance, in the example above the parser *satisfy is_digit* represents 10 different sentences, namely "0", "1" ,...,"9".

The definition of *terminal* appears now as a specialization of *satisfy*. Actually, *terminal* can be written in terms of *satisfy* easily:

```
terminal::A → parser [A] A
terminal T = satisfy (T==)
```

The new definition of *terminal* relies on $satisfy$ in order to get the representation, and therefore does not include the additional parameter.

We may also think that a good representation for parsers *terminal* and *satisfy* could be the own generated sentence $[T]$ instead of the element T. These variants of *terminal* and *satisfy* may be called *terminal_l* and *satisfy_l* respectively, denoting that they return lists, and may be defined as follows:

```
satisfy_l::(A → bool) → parser [A] A
satisfy_l P [R]      =   satisfy P R
terminal_l::A → parser [A] A
terminal_l T [R]     =   terminal T R
```

3.7 Parser Combinators.

We have already defined functions representing a few basic parsers. What we need now is a set of *Parser Combinators* in order to join the basic parsers in different ways and to enrich their expressiveness. The parser combinators are higher-order functions which take parsers as input parameters and return a new parser as output value. In the example at the figure 1 we have presented two such parsers: the operators // and ++. We need to adapt their definitions, as initially they did not deal with representations.

The choice operator // will be represented from now on through the non-deterministic function <|>. It takes two parsers and return one of them using non-determinism. Its representation is the representation of the selected parser:

```
infixr 10  <|>
(<|>)::parser A B → parser A B → parser A B
(<|>) P1 P2 R = P1 R
(<|>) P1 P2 R = P2 R
```

For instance the parser *alpha* generates all the strings whose only character is an alphanumeric character:

```
alpha::parser [char] char
alpha = (satisfy is_digit)  <|>  (satisfy is_letter)
```

where *is_digit* and *is_letter* are defined as convenient.

The updated version of the sequence operator ++ , which is denotated by <∗> , takes account of the representations. The representation of the second parser is supposed to be a list L of elements or type A, while the representation of the first one must be an element E of the same type A. The final representation is the list whose head is E and whose tail is L, i.e. the result of appending the representation of the second parser to the representation of the first parser:

```
infixr 20 <∗>
(<∗> )::parser A B → parser [A] B → parser [A] B
(<∗> ) P1 P2 [E | L] = (P1 E) ++ (P2 L)
```

The generated sentence is still the concatenation of the sentences generated from $P1$ and $P2$. Both <∗> and <|> associate to the right, and their priorities are defined as to minimize parentheses.

Sometimes the representation of one of the parsers in a sequence is not important (e.g. when it is a puntuaction mark), and we prefer to retrieve only the representation of the other parser. This variations of <∗> are the operators <∗ and ∗> defined as

```
infixr 20 <∗
(<∗ )::parser A B → parser C B → parser A B
(<∗ ) P1 P2 E = (P1 E) ++ (P2 _)


infixr 20 ∗>
(∗> )::parser A B → parser C B → parser C B
(∗> ) P1 P2 E = (P1 _) ++ (P2 E)
```

Another useful parser combinator usually defined in functional parsers is the parser that allows the repetitive application of a given parser. We call it the *star* combinator and admits the recursive definition:

```
star::parser A B → parser [A] B
star P = P <∗> (star P)  <|>  empty
```

that is, the sentences generated by *star P* are of the form $s_0 s_1 \ldots sn,\ n \geq 0$ being $s_1, s_2, \ldots s_n$ sentences generated by P. Of course, if P is a deterministic parser and its only generated sentence is s then $starP = \{s^n\ :\ n \geq 0\}$. Otherwise the s_i's of the previous expression may be any different sentence generated by P. The next example defines an identifier being a string of alphanumeric characters beginning with a letter:

```
ident::parser [char] char
ident = satisfy is_letter <*> star alpha
```

Using *star* it is easy to define the function *some* which allows the repetition of one or more sentences generated by the given parser. It is like *star* but discarding the empty word.

```
some::parser A B → parser [A] B
some P = P <*> (star P)
```

For intance, we could represent a non-empty sequence of letters a and b as:

```
sec_ab: parser [char] char
sec_ab = some (terminal 'a'   <|>   terminal 'b')
```

3.8 Handling non-determinism.

Although easily understable, the definition of *star* contains a quite subtle technical point which is worth noting. Actually, if we examine carefully the rule for *star* it seems erroneous. The problem is that in the right-hand side of the function the variable P appears twice, and due to *sharing* the two P's must stand for the same value. Thus, if the non-deterministic function P takes for example the values '0' and '1', then all the sentences generated using *star P* will be of the form "0^n" or "1^n", but they will not include, for example, "01".

To find out the answer to this apparently important drawback we have to examine where the non-determinism of P may come from. If we suppose that P is built using only the previous basic parsers and combinators, the only possible sources of non-determinism are parsers of the form (P1 $<|>$ P2), or *satisfy F*, being $P1$ and $P2$ parsers and F any boolean function. But examining again the definitions of $<|>$ and *satisfy* we may check that both functions *need the representation parameter to become concrete values.* This means that although $<|>$ and *satisfy* represent two or more sentences of the languages, they can not be directly reduced and will not become a concrete sentence until their representations are provided, or in other words, that they always appear as partial functions and not like concrete values. Thus, if P is a non-deterministic parser, the repetition of P in the body of *star* causes no trouble, as P is a partial function.

Conversely, the definition of *star* is reduced directly, as it doesn't need any parameters. Therefore, the different *program arity* of *star*, *satisfy* and $<|>$ introduces a sequence in the order the functions are reduced.

3.9 Functional-Logic do.

In the definition of the type *parser*, the output value of the parsers need to be a list, but any type is allowed for the representation. Anyway, each basic parser and basic parser combinator we have presented so far provides a specific *default representation*. For instance the parser *terminal* returns the parsed element as representation, while the parser *star P* retrieves a list with the sentences generated by using *P*. When defining more complicated parsers the user may provide its desired representation, but it is also possible forget representations and just rely on the default one.

For example, if we would like to parse senteces with one or more letters *a*, but we are not interested in the representation, we may just write:

```
as = some (terminal 'a')
```

and we may try the goal:

as R == "aaaa"

which returns *yes*, retrieving *R=="aaaa"* as default representation. Therefore if we don't care about representation we will get as default representation a copy of the output string.

This technique provides an useful way of taking care about representation when necessary, but dismissing it whenever the default representation seems appropiate. However, it also constraints the type of the parsers in order to match the type of the parameters of the combinators.

For example if we try the parser:

```
sec_abs: parser [char] char
sec_abs = (star (terminal 'a')) <*> (star (terminal 'b'))
```

the type checker returns an error, because the first argument of <∗> should be of type *char*, while *some (terminal 'a')* is of type *[char]*.

Of course we might define a wide set of combinators for sequence, extending the types to combine lists with lists. Another solution is to enforce all the representations to be lists which are easier to combine [4]. Any of these approaches lead to rather complicated parsers, partially loosing our aim of providing simple mechanisms for defining parsers easily. Moreover, the previous solutions do not tackle all the posibilities.

Consider that we would like to define a parser for *assign* sentences. This sentences are of the form form "Var=Expr;". The expressions are those we defined in figure 5, while the name of the variable is an *identifier* whose parser we defined above.

The representation of an assign sentence will be an element of the type:

```
data binding = string := prec_tree
```

where the string is the identifier of the variable, the value of type *prec_tree* is the representation of the expression as defined in figure 5, and ':=' is an infix data constructor.

It seems clear that parser *assign* should use parsers *expression* and *ident*. But the types of the representations of *expression* and *ident* are different, and

[4]This is done in next section.

neither <∗> nor the alternative operators we suggest above are able to combine the representations of these two parsers, yet the definition of *assign* we propose is meaningful.

The problem may be only accomplished defining an ad hoc sequence combinator, which combine elements of types *string* and *prec_tree*. Actually, what is needed here is a general frame for constructing such ad hoc sequence combinators. This is achived using the function *do*. It offers an alternative to the sequencing operator, without providing any implicit mechanism for building representations. The definition of the *do* construction is simply a generalization of our first version of the sequence operator, ++ :

```
do::[[A]] → [A]
do = concat
```

The *do* construction takes a list of parsers with their representations, that is a list of sentences of the same type, and just generates the sentence resulting of concat all the elements of its input list (*concat* is a standard function that concatenates a list of lists). This surprisingly simple construction provides a straightforward solution for the problem of defining the *assign* parser:

```
assign::parser binding char
assign (N := E) = do [id N, terminal '='_ , expression E, terminal ';' _]
```

The representation of each component parser appears in the form of argument, while the pattern $N := E$ is the ad hoc combinator we were looking for. Note the presence of anonymous variables _ to denote that we are not interested in the representations of the terminals '=' and ';'. We may try for example:

```
assign R == "Count=1*0+1;"
```

which succeeds with

```
R == "Count" := (op plus (op mult (val 1) (val 0)) (val 1))
```

Therefore the *do* construction provide an easy way of combining parsers, without the addition of any syntactic modification to the language.

As final examples of this subsection, the sequence combinators <∗> , <∗ and ∗> could have been easily defined using *do*:

```
(<∗> ) P1 P2 [H | T]   =   do [P1 H, P2 T]
(<∗ ) P1 P2 H          =   do [P1 H, P2 _]
(∗> ) P1 P2 T          =   do [P1 _, P2 T]
```

3.10 Limitations of the Generate and Test parsers.

In this subsection we will reach the limitations ot the parsers we have presented so far. These limitations will lead us to the *input/output parsers* whose detailed discussion is carried out in the following section.

So far, our parsers have retrieved representations using patterns built with the intermediate representations of its component parsers. However, the general definition of representations should consider combinating the intermediate representations using functions, not only through patterns. But this aim results, alas, not possible in the context of g&t parsers, as we show in the next example.

Consider that we would like to parse type declarations of the form

$$var_1, var_2, \ldots, var_n \ = \ type;$$

where the strings var_i represent the names of the variables, while the string *type* is the name of the type. As we prefer to relate each variable with its type, a possible representation for this sentence may be a list of pairs of strings , the first string of each pair representing the type name and the second string the variable name. A failed attempt of defining a parser satisfying these requirements is:

```
type_dec R   =   do [vars LVars, terminal '=' _, ident Type]
             ⇐   map (mkpair Type) LVars == R

vars         =   ident <*> (star (terminal ',' *> ident) )
```

The parser *vars* return a list of identifiers, using the parser *ident* defined previously, while the parser *type_dec* try to construct the list of pairs using the funcion *map*. The function *map* is defined as usual in functional programming:

```
map:: (A → B) → [A] → [B]
map F []       =   []
map F [X|Xs]   =   [(F X)|(map F Xs)]
```

The first argument of *map*, namely *mkpair Type*, is the function we pretend to apply to each element of the list, defined elsewhere as

```
mkpair :: A → B → (A,B)
mkpair X Y = (X,Y)
```

The variable *Type* is the name of the type, while the second argument of *map*, *LVars*, is a list whose elements are the names of the variables parsed by *vars*. If we try the goal:

```
type_dec R == "v1,v2=integer"
```

returns successfully de desired result

```
R == [ ("integer', "v1"), ("integer", "v2")]
```

The problem arises when we try to parse a sentence which is not valid. For example in the following goal the symbol *'='* after *v2* is missed:

```
type_dec R == "v1,v2integer"
```

the expected answer is *no* but, instead of that, the system just loops, without providing any answer.

To understand the reason of this strange behavior we must point out that the conditions of a function rule are computed *before* the body of the function. Thus, in the previous example the strict equality

```
map (mkpair Type) LVars == R
```

is performed at the start of the computation of the parser *type_dec*. At this point, the variables *Type*, *LVars* and *R* are unbounded. To solve the strict equality *map* first bound *LVars* to the empty list and the condition is satisfied. The *do* is then tried, but fails (the sentence "v1,v2integer" is not part of the language). Then backtracking enforces *map* to take another alternative and it bounds *LVars* to a list with at least one element. The *do* is then tried again, and of course fails again. Repeating the actions, the strict equality of the condition will be satisfied with an infinite number of lists, while the construction *do* always will fail, and hence the goal never fails nor succeeds.

This behaviour may be generalized to several other situations in which we need to evaluate a general expression to construct the representation. In fact, if we use conditions to build representations, we must keep in mind that the variable standing for the intermediate representations are unbounded when evaluating the conditions, and hence we must assure that the expression involved are deterministic in these conditions. In the example above this rule is not satisfied, as function *map* behaves in a non-deterministic way if its second argument is an unbounded variable. On the other hand, notice that patterns are always deterministic, regardeless of the intantiation of the variables, and therefore can be used safely.

Unfortunately, this important drawback of g&t parsers is not avoidable. Therefore these parsers must be used mainly to build a kind of intermediate representations through pattern expressions, which can be processed later. Of course, this point of view complicate the parsing process, for now is necessary a second stage that converts the intermediate representation into final values. Hence, it would be desiderable that parsers could retrieve the final representations directly. This aim is satisfied by the *input/output parsers*, which are presented in the next section.

4 Input/Output parsers.

4.1 Input/Output Parsers.

In this section we introduce a new kind of parsers. Conversely to the g&t parsers, the new ones will have the input sentence as input parameter. If some prefix of the input sentence is recognized as part of the language, the remainder list, i.e. the yet not parsed part of the sentece, is returned as output value.

Therefore, the type of the *input/output parsers*, or simply *parsers* may be defined as:

$$\text{type parser Repr Token} = \text{Repr} \rightarrow [\text{Token}] \rightarrow [\text{Token}]$$

Of course, the previous type definition is not the only one feasible (see the discussion of the type for the g&t parsers), but we have adopted the definition above as it allows building the representations in the apealling way showed in the previous section. In addition, the use of the extra argument will be decisive in a later section, where the use of parsers as data values is discussed.

4.2 Basic Parsers and Parser Combinators,

Carrying the input sentence as an explicit parameter enforce basic parsers and parser combinators to perform the actions that were performed by the system in the case of the g&t parsers. For instance, the function *satisfy* need to remove the recognized character from the input string, returning the rest as output value. The complete set of redefined functions may be seen in figure 6.

```
type parser Repr Token = Repr → [Token] → [Token]
empty:: parser [A] B
empty [] L           =   L

satisfy:: (bool → A) → 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 6: Basic parsers and parser combinators.

Observe that the definitions are very similar to those of the previous section, but including some treatment for the input sentence. This is particularly obvious in the case of the sequence combinator. In this case the first condition is used to apply the first parser and get its output sentence *O1*, which is the input parameter of the second parser as showed in the body of the function. The representation is the same that in the case of g&t parsers. Furthermore the properties presented in the previous section - the use the parsers as generators, the skill in defining context sensitive grammars - are also valid for input/output parsers.

Note, however, that the representations of *terminal* and *satisfy* have changed, as they now return a list with a single element - the recognized symbol - instead of the symbol alone. This is due to a technical reason, namely that it provides uniformity to the parser definitions, as now all the basic parsers return lists. Actually, they could have been defined in this way for the g&t parsers, but the approach considered there was useful for constructing representations as patterns, while the one considered in this section provides a reduced set of combinators. The problem of combining representations of different types will be tackled by the new definition of the *do* combinator.

Al the other combinators defined in the previous sections, as *star*, *some*, *terminal_l* or *alpha* are still valid, as a nice outcome of the new definitions.

4.3 The *do* family.

For the moment, the new parameter only has complicated the definitions of the basic parsers and parser combinators, providing no visible benefit. But this apparent nuisance becomes useful when constructing involved representations. Effectively, the function *do* presented in the previous section only was used when constructing representations as patterns. Here, the function *do* will be used to construct general representations, that is, representations given by a expression, usually depending upon the representations returned by the component parsers. Remember that this was not possible using g&t parsers, and hence this is the major contribution of the input/output parsers.

We first define a version of do, what we call simple do or *do_s*, which simulates the g&t *do*. Now the definition is slightly more involved, as we must carry the input sentence throughout the computation.

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

The type *[[A]→ [A]]* corresponds to a list of parsers with their explicit representations. The definition of *do_s* applies the input parsers in sequence, passing the output string of a given parser as the input string of the next one. This function can replace the g&t *do* with the same behaviour.

Following, the definition of the new *do* is introduced. It takes two additional parameters, the expression with the representation of the *do* sequence and the representation parameter of the function . If all the parsers in the list succeed, the expression is evaluated and tried to match with the representation of the function.

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

In order to understand the possibilities provided by this definition, we revise the example that showed the limitations of the g&t parsers, that is the definitions of multiple identifiers. The definition of *type_dec* is now

```
type_dec R    =   do_s [vars LVars, terminal '=' _, ident Type]
              ⟸   map (mkpair Type) LVars == R

vars          =   ident <*> (star ((terminal ',') *> ident) )
```

where

```
ident = satisfy is_letter ++ star (satisfy is_digit  <|>  satisfy is_letter)
```

and

```
(*> ) P1 P2 T = do_s [P1 H, P2 T]
```

Again, if we try the goal

```
type_dec R == "v1,v2=integer"
```

the parser succeeds, but the goal

```
type_dec R == "v1,v2integer"
```

does not end, although it should fail, due to the reason explained in the previous section. Fortunately, now we have the *do* combinator. Simply defining *type_dec* as

```
type_dec R = do   [vars LVars, terminal '=' _, ident Type]
                  (map (mkpair Type) LVars) R
```

avoids the problem, and now the second goal returns simply *no*.

As a final member of what we have called the *do family*, we present the infix operator $\longrightarrow$. The expression $P \longrightarrow R$ is the parser that recognizes the same sentences as P but whose representation is R. This combinator is useful when we pretend to change the default representation provided by a parser by a different one, and will be used in the new version of the parser for expressions we present next.

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

4.4 Expressions again.

Our 'classical' example of arithmetic examples is here revisited. Using the *do family* of combinators we can define a parser for expressions whose representations are the result of evaluating the expression. Furthermore, the grammar is extended to work with integer numbers, not with just the values '0' and '1'. The parser is presented in figure 7.

```
expression          =   term   <|>   plus_minus_expr
plus_minus_expr R   =   do [term T, plus_minus Op, expression E] (Op T E) R
term                =   factor   <|>   prod_div_expr
prod_div_expr R     =   do [factor F, prod_div Op, term T] (Op F T) R
factor              =   num   <|>   par_expr

par_expr R          =   do_s [terminal '(' _, expression R, terminal ')' _]
num R               =   do [some digit L] (numeric_value L) R

numeric_value L     =   foldl ((+).(10*)) 0 (map val L)
digit               =   satisfy is_digit
plus_minus          =   (terminal '+')⟶ (+)   <|>   (terminal '-')⟶(-)
prod_div            =   (terminal '*')⟶(*)   <|>   (terminal '/')⟶(/)
```

Figure 7: Parser recognizing and evaluating expressions.

The representation of parsers *plus_minus* and *prod_div* is now the infix operator they represent, which is applied when building the representations of *plus_minus_expr* and *prod_div_expr*. The representation of a number is its numeric value, which is returned by the function *numeric_value*. This function used the standard functions *map* and *foldl*, together with a function *val* that transforms a single digit into its numeric value. As an example, the goal

```
expression R "((10+4)*20-30)/50" == []
```

succeeds with

```
R == 5
```

4.5 Limitations of *do*.

It is very important noting the argument *R* in the definition of *num*. It stands for the variable that finally contains the representation of the number and *must not be removed of the definition of the parser*. The reason is related to the function *star* and its shared variable *P*, as discussed in Chapter 3. Without the representation parameter, the parser *num* could be directly reduced to the *do* construction and, due to the explicit representation *L*, the parser *star num* would recognize only repetitions of the same number (that is, numbers with the same representation *L*).

Thus, to avoid involved problems, if we are going to use the function *star*, we should not use constructions *do* inside a *star*, neither directly nor through intermediate functions without arguments. This can be avoided including always the parameter *R* in the definition of functions with *do* constructions.

5 Numerical Constraints.

The growing interest in languages representing spatial relationships (e.g. visual languages [HMO91]) has introduced the study of *numerical constraints* in relation to the parsing problem. In this section we show a very simple but suggestive example of how our parsers can integrate numerical constraints easily. The usefulness of non-determinism in this area is also sketched.

Supposse we are interested in a parser for recognizing *boxes*. The terminals of the language will be pairs of integers representing points in the plane, and a valid sentence will be a sequence of four points standing for the corners of the box, beginning with the lower-left and following anti-clockwise. The desired representation is a pair of points representing the lower-left and the upper-right corners of the box.

```
box     R =  do [point (X1,Y1), point (X2,Y2), point (X3,Y3), point (X4,Y4)]
               ((X1,Y1),(X3,Y3))  R
        <==  Y1==Y2, X1==X4, X2==X3,Y4−Y1==Y3−Y2, Y1<Y4, X1<X2
point   R =  do [terminal (X,Y) _ ] (X,Y) R
```

The conditions assure that the points actually represent a box. Note that these constraints are settled before parsing the points. As a consequence, if the points do not represent a box, the parser can fail as soon as possible. For instance, if the condition *Y1==Y2* is not verified, the parser will fail just after parsing the second point.

For our example to work properly, the language must be able to handle numerical constraints concerning still uninstantiated variables, and to incrementally check the satisfactibility of the accumulated constraints whenever new ones are imposed during the computation. Such an extension of the language considered so far is described in [JM+92], and is actually implemented in the system $\mathcal{TOY}$ (with such this example is indeed executable).

Supposse now that we would like to recognize boxes, without concerning about the positions of the points in the list. This means that four points form a box if there exists a permutation of the points that is recognized by parser *box*. We may define the non-deterministic function *permut*, which relies in the non-deterministic function *insert*:

```
permut []        =  []
permut [X|Xs]    =  insert X (permut Xs)

insert X []      =  [X]
insert X [Y|Ys]  =  [X,Y|Ys]
insert X [Y|Ys]  =  [Y|insert X Ys]
```

and try *box* with a non-ordered set of points

```
box R (permut [(60,80),(10,80),(60,20),(10,20)]) == []
```

which returns successfully

```
R == ( (10,20), (60,80) )
```

This solution may seem rather awkward, but actually, due to laziness, it is quite efficient. Indeed, we have seen that the conditions assure that the parser *box* fails as soon as possible. This means that if a permutation is not valid, *permut* will generate only the elements neccessary to make the parser fail, not the whole permutation. Then, *permut* will try the next possibility using backtracking, but discarding only the element that has triggered the fail, preserving the part of the permutation that can be still valid. Therefore, the solution, if exists, is found out using a clever strategy that minimizes the number of failed attempts. It is worth noticing that function *permut* used in this way is another example of the FLP *generate and test* technique, which we used before to define the g&t parsers.

6 Parsers as data.

6.1 Canonical Definitions.

In previous subsections the advantages of defining parsers using non-deterministic FLP have been discussed. Here we intend to show how functional-logic languages that include *higher-order patterns* can consider parsers as truly *first class data values*. To achieve our objective, we need to restrict the shape of our parsers, according with the next definition.

We say that a parser p is defined *canonically* iff its definition consists in a single rule adopting one of the following forms:

(i)	p R = empty R	(ii)	p R = terminal R
(iii)	p R = (p1 ◇ p2) R	(iv)	p R = star p1 R

where *p1* and *p2* are canonicallly defined, and $\diamond \in \{$<∗> ,<|>$\}$. In the rest of the subsection we consider only parsers defined canonically, which we call *non-terminals*. Observe that these parsers are expressive enough to represent any set of BNF-rules and therefore any context-free grammar.

Functions *empty*, *terminal*, <∗> , <|> and *star* are the only basic parsers and parser combinators allowed, and from now on we call them the *basic set*. Their definitions are those presented in the previous section, excepting the definition of *star*, which is:

```
star:: parser [A] A → parser [A] A
star Rep Input = (P <∗> (star P)   <|>   empty) Rep Input
```

The inclusion of the two explicit arguments, *Rep* and *Input*, is necessary for reasons that will become apparent soon. Note that this new version of function *star* could have been introduced in the previous section as it was valid also in that context.

As an example to be used in this section, parser *list* recognizes lists of 0's and 1's separated by commas.

```
list        R =   (terminal '[' <∗> body_list <∗> terminal ']') R
body_list   R =   (element <∗> star (terminal ',' <∗> element)   <|>   empty) R
element     R =   (terminal '0'   <|>   terminal '1') R
```

Here *list*, *body_list* and *element* are non-terminals.

6.2 Handling Productions.

The key for handling parsers as data is the skill in 'exploring' right-hand sides of productions (i.e. parser functions) provided by HO patterns. The first step in this direction is function *get_right* which retrieves right-hand sides of productions corresponding to non-terminals in the grammar.

```
get_right::parser A B → parser [B] B
get_right P = contract (expand P)

expand:: parser A B → ( [B] → [B])
expand P = P _

contract:: ([A] → [A]) → parser [A] A
contract (empty _)            =   empty
contract (terminal A _)       =   terminal A
contract ( (<∗>) P1 P2 _)     =   P1 <∗> P2
contract ( (<|>) A B _)       =   P1 <|>P2
contract ( star P _)          =   star P
```

To get the right-hand side of the parser *P* is used the function *expand*, which returns the result of applying the parser to an arbitrary argument '_'. Effectively, every non-terminal is reduced to its right-hand side when applied to a single argument, for it must have any of the shapes described in *(i)-(iv)*. For the same reason, the outermost function of a right-hand side belongs to the basic set, and all of the functions in the basic set need two arguments to be reduced (that is the reason for adding the two extra arguments to *star*). To show this, we may try for intance the goal

```
expand list == L
```

which succeeds with the value

```
L == (terminal '[' <*> body_list <*> terminal ']') _
```

Thus, function *expand* allows getting the right side of a non-terminal partially applied, that is, including an extra argument (for this reason it returns a *parser_rec* instead of a *parser*). Function *contract* performs the reverse action: it removes the extra argument from a partially applied function of the basic set. Here the use of HO patterns is essential, as this technique allows considering partially applied functions as constructed data, with the name of the function regarded as the data constructor.

Now the meaning of *get_right* becomes evident. For example the goal

```
get_right list == L
```

succeeds with

```
L == terminal '[' <*> body_list <*> terminal ']'
```

while

```
get_right empty == L
```

returns

```
L == empty
```

because *empty* is not a non-terminal.

The difference between non-terminals and the rest of the valid parsers is pointed out by the boolean function *nonterminal*.

```
nonterminal P = not (get_right P == P)
```

Also, based on the definition of *get_right* is easy to define the boolean infix operator >>. The expression $L >> R$ is true if R is the definition of the non-terminal L:

```
(>>):: parser [A] A → parser [A] A → bool
L >> R = true ⇐= R == get_right L, not (R == L)
```

The first condition assures that R is the right-hand side of L, while the second one checks whether L is actually a non-terminal or not.

6.3 First and Follow.

Two very important sets related to the underlying grammar are that of the terminals and non-terminal symbols of the grammar. Function *symbols* takes a non-terminal S representing the initial symbol of the grammar and returns a pair of lists representing the terminal and non-terminals reachable from S.

```
symbols :: parser [A] B → ([B], [parser [A] B])
symbols P                       =  get_symbols P ([ ],[ ])

get_symbols :: parser [A] B → ([B],[parser [A] B]) → ([B], [parser [A] B])
get_symbols empty L             =  L
get_symbols (terminal A) (T,NT) =  if A 'in' T then (T,NT) else ([A|T],NT)
get_symbols (A <*> B ) L        =  get_symbols B (get_symbols A L)
get_symbols (A <|>B) L          =  get_symbols B (get_symbols A L)
get_symbols (star P) L          =  get_symbols P L
get_symbols P (T,NT)            =  if P>>R then
                                     if P 'in' NT then (T,NT)
                                     else get_symbols R (T,[P|NT])
```

Most of the work is performed by function *get_symbols*. It takes the parsers that are currently being examined, and a pair of lists with the intermediate results. The purpose of these lists is to avoid the inclusion of repeated elements in the final lists, and - more important - to elude possible infinite loops. This is done in the last rule, which is devoted to get the symbols when the parser is a non-terminal. In this case, the first condition $P >> R$ is used to get the right hand-side of P, checking at the same time if P is actually a non-terminal ($>>$ would fail otherwise). The second condition *if P 'in' NT* checks whether the current non-terminal has been examined yet, and it is at this point where the possible infinite loop is broken. Without this condition the goal

```
symbols list == S
```

would loop, but conversely it returns:

```
S == ( "],10[", [ element, body_list, list ])
```

which are the five terminals and the three non-terminals accessible from *list.* The function *'in'* used to check if an element is part of a list can be defined easily as:

```
X 'in' [ ]     =  false
X 'in' [Y|Ys]  =  if X==Y then true else X 'in' Ys
```

The rest of the rules only collect symbols from parsers whose outermost function are in the basic set, that is from parsers that do not represent non-terminals but right-hand sides.

Function *get_prods* collects every production accessible from a non-terminal P. To represent productions pairs of values of type *parser* are employed, where the first component stands for the left hand-side and the second one for the right hand-side of the production, i.e. of the function rule.

```
type prod A = (parser [A] A, parser [A] A)
get_prods::parser [A] A → [prod A]
get_prods P = zip N (map get_right N) ⟸ symbols P == (_,N)
```

At this point we can define the functions neccessary for checking, for intance, whether a given grammar satisfies the $LL(1)$ property. Although there are a number of efficient algorithms for checking the property directly, we are going to follow the steps enumerated in ([ASU86]), which provide some interesting intermediate functions.

The properties presented from now on need an extended definition for the terminals of the grammar, including the empty word and the terminal representing the end of the input string. Therefore we introduce the following data type:

```
data extended_symbols A = empty_word | term A | eos
```

That is, the set of extended symbols include all the terminals of type A through the data constructor *term*, plus the values - constructors of arity 0 - *empty_word* and end of string (*eos*).

We begin by defining the function *first*, which takes a non-terminal P as input parameter and returns the set of terminals that can start some sentence derived from P. If the empty word can be derived from P, it is also returned.

```
first :: parser [A] A → [extended_symbols A]
first empty          =   [empty_word]
first (terminal A)   =   [term A]
first ( A <*> B)     =   if empty_word 'in' L then L 'union' (first B) else L
                         ⟸ first A == L
first (A <|>B)       =   (first A) 'union' (first B)
first (star A)       =   [empty_word | first A]
first P              =   if P>>R then first R
```

Given a non-terminal as input parameter, the first rule applied is the last one, which says that the set of first symbols of a non-terminal P is the set of first symbols corresponding to its right-hand side. The condition *P>>R)* assures that P is actually a non-terminal. Among the other rules, perhaps the only one that deserves explanation is that of the sequence combination of two parsers A and B. The condition is here used to avoid repeated computations. It bounds the new variable L to the set of *first* symbols of A. Then, if the empty word is not in the set, the first symbols of the sequence are precisely those in L. Otherwise the first symbols of B can appear also at the first position of some sentence generated by the sequence, and we must return the union of the two lists. Function *union* concats the elements of two lists avoiding repetitions:

```
union [] L       =   L
union [X|Xs] L   =   if X 'in' L then union Xs L else [X|union Xs L]
```

To check if a given grammar satisfies the *LL(1)* condition we need the *first* sets of every non-terminal in the grammar. We call this function *all_first*:

```
type symbols_assoc A = [(parser [A] A, [extended_symbols A])]
all_first:: parser [A] A → symbols_assoc A
all_first P = zip N (map first N) ⟸ symbols P == (T,N)
```

The type definition *symbols_assoc* represents the set of extended symbols associated to any non-terminal in the underlying grammar. It will be used in several ocassions in the rest of the section. The standard functions *map* and *zip* are used here to apply the function *first* to all the non-terminals and to match every non-terminal with its list of *first* symbols, respectively. For example:

```
all_first list == F
```

returns

```
L == [   (element,     [ (term '0'), (term '1') ]),
         (body_list,   [ (term '0'), (term '1'), empty_word ]),
         (list,        [ (term '[') ])   ]
```

The next function defined is *follow* which returns a list of pairs *(NT, F)*, where *NT* is a non-terminal and *F* is the set of terminals that can appear immediately to the right of *NT* is some sentencial form. To calculate each set of *follow* symbols we consider the next rules:

1. The special symbol *eos* is a *follow* symbol of the start non-terminal of the grammar.

2. If there is a production of the form $NT \overset{*}{\Longrightarrow} \alpha <\!*\!> A <\!*\!> B$ with A a non-terminal, then everything in *first B*, excepting the empty word is in the set of *follow* symbols of A. If the empty word is in *first B* then everything in *follow NT* is also in *follow A*.

3. If there is a production of the form $NT \overset{*}{\Longrightarrow} \alpha <\!*\!> A$ with A a non-terminal then all the symbols of *follow NT* are also symbols *follow A*.

Here is the definition of *follow* together with some definitions of its auxiliar functions.

```
follow :: parser [A] A → symbols_assoc A
follow P          =    follow' [(P,[eos])] (get_prods P)

follow'::symbols_assoc A → [prod A] → symbols_assoc A
follow' List N    =    if List==List' then List else follow' List' N
                       ⟸ foldl follow_prod List N == List'
```

Function *follow* gets the start non-terminal of the grammar *P* and collect all the accessible productions. It also includes *eos* as a follow element of *P*, observing the rule 1. Function *follow'* call function *follow_prod* once and again until no more follow symbols are discovered.

The next function, *follow_prod*, examines a fixed production and updates the table of *follow symbols*, adding the new information entailed from the structure of the production.

```
follow_prod List (_,empty)                                =   List
follow_prod List (_,terminal T)                           =   List
follow_prod List (NT,(<*> ) A B)                          =
      if nonterminal A
      then follow_nonterm NT List A B
      else follow_prod (follow_prod List (NT,A)) (NT,B)
follow_prod List (NT,star A)                              =
      follow_prod List (NT,A <*> A  <|>  empty)
follow_prod List (NT,(<|>) A B)                           =
      follow_prod (follow_prod List (NT,A)) (NT,B)
follow_prod List (NT,A)                                   =
      if nonterminal A
      then (add_symbols A (get_index NT List) List )
```

If the production is of the shape *NT R = empty R* or *NT R = terminal T* then no additional symbols are discovered and the resulting list is kept unaffected. If it is a sequence of parsers, and the first parser is a non-terminal, then the function *follow_nonterm* is called. This function regards the rule 2 to get new follow symbols.

```
follow_nonterm P List A B =
      if empty_word 'in' FirstB
      then follow_prod (add_symbols A (get_index P List1) List1 ) (P,B)
      else follow_prod List1 (P,B)
    <== first B == FirstB,
        add_symbols A (filter (/= empty_word) FirstB) List == List1
```

In the case of a sequence whose first component is not a non-terminal, structural induction is performed by *follow_prod*. Similarly, in the case of alternative of parsers, both components are examined and their results mixed in a single list. A production of the form *NT R = star P R* is regarded as equivalent - in terms of the set of follow symbols - to a production of the form *NT R = (P <*> P <|>empty) R*. Finally the last rule of *follow_prod* considers the possibility sketched in rule 3.

The function *add_symbols* includes new symbols in the follow set of a non-terminal, while *get_index* retrieves the current follow set associated to a given non-terminal.

```
add_symbols A NewS []            =   [(A,NewS)]
add_symbols A NewS [(E,Sym)|R]   =   if A==E
                                     then [(E, Sym 'union' NewS)|R]
                                     else [(E,Sym)|add_symbols A NewS R]

get_index NT []                  =   []
get_index NT [(A,R)|Xs]          =   if NT==A
                                     then R
                                     else get_index NT Xs
```

As an example, the goal

```
follow list == L
```

returns

```
L == [ (list, [ eos ]), (element, [ (term ','), (term ']') ]), (body_list, [ (term ']') ]) ]
```

6.4 Constructing the Predictive Parsing Table.

The next function, and the last step before defining the function that checks if the grammar has the *LL(1)* property, is to define the *predictive parsing table* of the grammar whose initial symbol is represented by the parser *P*. If the grammar has *n* different accessible non-terminals and *m* accessible terminals, the predictive parsing table will have *n* rows and *m+1* columns (the extra column for the special symbol *eos*). Each entry *(NT,T)* of the table is a list with the productions that may be tried if we are reducing the non-terminal *NT* and the leftmost terminal of the input sentence is *T*. For instance the entry *(body_list, ']')* of the table corresponding to the grammar of the list expressions will containt only the production *empty* while the entry *(body_list, '1')* will contain *element <*> star (terminal ',' <*> element)*. Of course the grammar is *LL(1)* if and only if all the entries of its predictive parsing table have one production at most.

In order to construct the predictive parsing table, is useful to describe the ideas in terms of grammar productions instead of function rules. We also use the notation $FIRST(\alpha)$ and $FOLLOW(A)$ to denote the set of *first* symbols of a sentencial form α and the set of the *follow* symbols of a non-terminal, respectively.

Suppose $A \rightarrow \alpha$ is a production of the grammar with $a \in FIRST(\alpha)$. Then, the entry (A, a) of the table must contain the production $A \rightarrow \alpha$. The only complication occurs when the empty word is in $FIRST(\alpha)$, then we can reduce A also when the input sentence begins by any symbol $b \in FOLLOW(A)$ and therefore the entry (A, b) must contain also $A \rightarrow \alpha$. The code of the function *parsing_table*, is

```
parsing_table ::parser [A] A → [ [ [parser [A] A] ] ]
parsing_table P =  foldl (parsing_table_prod Follow (T',N)) Table Prods
              ⇐   symbols P == (T,N),
                   T' == [eos|map term T],
                   iniTable (length N) (length T') == Table,
                   get_prods P == Prods,
                   follow P == Follow
```

The type of the function shows that it gets a parser (actually a non-terminal) and retrieves a list of lists of lists. The first level of nesting represents the rows of the table, and each element of a row, i.e. each column, is a list of parsers. Actually it should be a list of productions, but we only store the parser corresponding to the right hand-side of the production, for the left hand-side, the non-terminal, is determined by the index of the corresponding row. The order between non-terminals is determined by the order in the list returned by function *symbols*, and the same is valid for the terminals.

The conditions of the function first of all get the set of symbols of the grammar and constructs the empty table, using the function

```
iniTable LengthN LengthT = take LengthN (repeat (take LengthT (repeat [])))
```

The dimensions of the table are the length of the list of non-terminals and the length of the list of terminals including *eos*. The set of *follow* symbols is also collected in order to elude its repetitive calculation. Finally, the body of *parsing_table* uses the standard function *foldl* to examine all the productions, building incrementally the final table. The function that examines each production is *modify_parsing_table*, but the function *parsing_table_prod* is used as a previous filter, splitting the productions of the form $A \leftarrow \alpha <|>\beta$ in two productions $A \leftarrow \alpha$ and $A \leftarrow \beta$, assuring that finally function *modify_parsing_table* gets the productions free of alternatives.

```
parsing_table_prod Follow Symbols Tab (NonTerm,((<|>) A B))    =
      parsing_table_prod Follow Symbols
           (parsing_table_prod Follow Symbols Tab (NonTerm,A)) (NonTerm,B)
parsing_table_prod Follow Symbols Tab (NonTerm, P)             =
      if (P==empty) ∨ (P==terminal _) ∨ (nonterminal P) ∨
         (P == (star _)) ∨ (P == (_ <*> _))
      then modify_parsing_table Follow Symbols Tab (NonTerm,P)
```

Therefore *modify_parsing_table* gets a production without alternatives and include the new information in the table.

```
modify_parsing_table Follow Symbols Tab (NonTerm,P) =
        if empty_word 'in' FirstP
        then add_list_parsing_table Symbols P NonTerm TabFirst FollowNonTerm
        else TabFirst
        ⟸
        first P == FirstP, NonEmptyFirstP==filter (/=empty_word) FirstP,
        get_index NonTerm Follow == FollowNonTerm,
        TabFirst == add_list_parsing_table Symbols P NonTerm Tab NonEmptyFirstP
```

The conditions include the right hand-side P of the production in all the entries of the form *(NonTerm,a)*, with *a* terminal in the set of the *first* symbols of *P*. This new table is called *FirstP*, and, if *empty* is not if the set *FIRST(P)*, is the value returned. Otherwise, we need to include the production also for the terminals in the set of the follow symbols of *NonTerm*, as is done in the *else* part of the body.

Finally the following functions include the information in the table.

```
add_list_parsing_table Symbols P NonTerm Tab List =
        foldl (add_parsing_table Symbols P NonTerm) Tab List

add_parsing_table (T,N) New NonTerm Tab Term =
        if New 'in' SetTerm then Tab
        else PrefNon++[PrefTerm++[[New|SetTerm]|PostTerm]|PostNon]
 ⟸
        pos T Term 1 == PosTerm,
        pos N NonTerm 1 == PosNonTerm,
        splitAt (PosNonTerm-1) Tab == (PrefNon,[LNonTerm|PostNon]),
        splitAt (PosTerm-1) LNonTerm == (PrefTerm, [SetTerm|PostTerm])

pos [X|Xs] E N = if X==E then N else pos Xs E (N+1)
```

Now we are ready to get the predictive parsing table of the grammar whose initial symbol is *list*.

```
parsing_table list == T
```

the result returned appears rather tangled:

```
T == [   [ [], [], [], [ (terminal '1') ], [ (terminal '0') ], [] ],
         [ [], [ empty ], [], [ element <*> (star (terminal ',') <*> element) ],
              [ element <*> (star (terminal ',') <*> element) ], [] ],
         [ [], [], [], [], [], [ (terminal '[') <*> body_list <*> (terminal ']') ] ]    ]
```

but can be interpreted if we remember the set of symbols associated to this grammar.

```
Symbols == ( "],10[", [ element, body_list, list ])
```

That is, the first row corresponds to *element*, the second one to *body_list* and the third one to *list*, while the six columns correspond to the right hand-side of the productions that we can choose if the input sentence begins with *eos*, *']'*, *','*, *'1'*, *'0'* and *'['* respectively. Specifically, the fourth element of the first list, shows that if a terminal *'1'* is found while reducing the non-terminal *element*, then it must be reduced using *terminal '1'*. It is worth noting that the empty lists correspond to entries of the table with no possible production to choice, i.e. erroneous derivations.

6.5 Defining the LL(1) Property.

Now the definition of the function *ll_1* is straightforward, for it only needs to assure that there is no entry (Noterm,Term) with two or more possible choices.

```
ll_1 :: parser [A] A → bool
ll_1 P = filter ((>=2).length) (concat (parsing_table P)) == []
```

The next goal succeeds with *R==yes*, showing that the grammar used in this subsection as example has the *LL(1)* property.

```
ll_1 list == R
```

A famous grammar that does not satisfy the *LL(1)* property is that of the if sentences with optional *else*s (known as the problem of the *dangling else*, see [ASU86]).

The problem arises because in the sentence "ixtixtoeo" - **i**f e**x**pression **t**hen **i**f e**x**pression **t**hen **o**rder **e**lse **o**rder - the grammar does not specify if the *else* part corresponds to the first *if* or to the second one. Therefore the goal

```
ll_1 sentence == R
```

returns

```
R == false
```

Moreover, by examining the parsing table, we can discover the entry with two possibilities for the table returned by

```
sentence R     =   ( terminal 'i' <*> expression <*> terminal 't' seqp sentence <*> else_part
                   <|>
                   terminal 'o' ) R

else_part R    =   (terminal 'e' <*> sentence   <|>   empty) R
expression R   =   (terminal 'x') R
```

Figure 8: parser for *if* sentences with optional *else* parts

```
parsing_table sentence == T
```

is of the shape:

```
T == [   [ [ empty ], [], [ empty, (terminal 'e') <*> sentence ], ...
         .
         .
         .
     ]
```

And the set of symbols returned by function *symbols* is

```
Symbols == ( "oetxi", [ else_part, expression, sentence ])
```

meaning that the first row corresponds to the non-terminal *else_part*, while the third column corresponds to the second terminal (the first column is for *eos*), which is *'e'*. Therefore the complication arises when trying to reduce *else_part* with *'e'* at the begin of the input sentence.

7 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* contruction) resembling FP monads in the style of parsers that can be written, but with the advantage of no needing any extra syntactic support. As another interesting contribution, we show how the inclussion of higher-order patterns in the language provides a new dimension to FLP parsers, as they can be regarded as truly, manageable, first-class data values.

There are many aspects of functional-logic parsers that deserve a thorough study, as they could yield new interesting outcomes. One of these aspects is the application of the g&t technique to other areas where rather involved search are needed. In this situations, the g&t technique may define solutions close to the high-level specifications, providing a more abstract framework. Another possibility, simply sketched in the paper, is the application of non-deterministic FLP parsers with numerical constraints for parsing visual languages. Indeed, the characteristics of these parsers might provide a suitable framework for parsing such languages. Finally, this discussion also suggests a FLP technique used as alternative to *monads*, and may be worth studiyng whether such solution can be generalized to other areas where *monads* have been employed successfully.

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

[Ant92] S. Antoy. *Definitional Trees*, In Proc. ALP'92, Springer LNCS 632, 1992, 143–157.

[AGL94] P. Arenas-Sánchez, A.Gil-Luezas, F.J. López-Fraguas. *Combining Lazy Narrowing with Disequality Constraints.* Procs. of PLILP'94, Springer LNCS 844, 385–399, 1994.

[AH+96] P. Arenas-Sánchez , T. Hortalá-González,F.J. López-Fraguas, E. Ullán-Hernández. *Functional Logic programming with Real Numbers*, in M. Chakavrarty, Y. Guo, T. Ida (eds.) *Multiparadigm Logic Programming*, Post-Conference Workshop of the JICLP'96, TU Berlin Report 96-28, 47–58, 1996.

[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.

[Col78] A. Colmerauer: *Metamorphosis Grammars*, in L. Balc (ed) *Natural Language Communication with Computers*, Springer, 1978,133–189.

[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.

[GHR93] J.C. González-Moreno, T. Hortalá-González, M. Rodríguez-Artalejo. *On the Completeness of Narrowing as the Operational Semantics of Functional Logic Programming.* Procs. of CSL'92, Springer LNCS 702, 216–230, 1993.

[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.

[HMO91] R. Helm, K. Marriot and M. Odersky. *Building Visual Languages Parsers.* ACM CHI'91. ACM Press 1991, pp. 105-112

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

[JM+92] J. Jaffar,S. Michaylov,P.J. Stuckey, Yap R.H.C. *The CLP($\mathcal{R}$) Language and System.* ACM Transactions on Programming Languages and Systems, Vol. 14, No. 3, 339–395, July 1992.

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

[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 *Some Applications of Functional-Logic Programming.* in M. Chakavrarty, Y. Guo, T. Ida (eds.) *Multiparadigm Logic Programming*, Post-Conference Workshop of the JICLP'96, TU Berlin Report 96-28, 35–45, 1996.

[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.

[War82] D.H.D Warren. *Higher-order extensions to Prolog: are they needed?*. J.E. Hayes J.E.,D. Michie D. and Y-H. Pao (eds), *Machine Intelligence 10*, Ellis Horwood, 441–454, 1982.