Institute for Perception Research
P.O. Box 513, 5600 MB Eindhoven

RL/jg 93/07
08.09.1993

Rapport no. 927

Formulating parsing theory with bunch theory

R. Leermakers

(a 'journal version' of manuscript no. 909)

Formulating Parsing Theory with Bunch Theory

René Leermakers

(Institute for Perception Research

P.O. Box 513, 5600 MB Eindhoven

email: leermake@prl.philips.nl)

Abstract

Much of mathematics, and therefore much of computer science, is built on the notion of sets. In this paper it is argued that in computer science it is sometimes convenient to replace sets by a related notion, *bunches*. The replacement is not so much a matter of principle, but helps to create a more concise theory. Advantages of the bunch concept are illustrated by using it in descriptions of functional parsing algorithms and attribute grammars.

I. INTRODUCTION

Parsers are often defined only for the subclass of context-free grammars for which they are deterministic. Mathematically, these subclasses, such as LL and LR grammars, are rather artificial compared to the entire set of context-free grammars. This makes deterministic parsing algorithms less amenable to mathematical methods than their nondeterministic generalizations. This is especially true if the generalization is described functionally. Thus, it is generally wise to first develop a nondeterministic functional parser for arbitrary grammars and to specialize to a subclass afterwards.

If a nondeterministic parser is applied to a grammar for which it is not deterministic, it is of little use, however. For instance, if an LR parser encounters a shift-reduce conflict, it may choose 'shift' where it should have 'reduced'. Angelic processes, ones that make only correct choices, do not exist in the real world. Thus, in order to be useful, a nondeterministic parser should be changed into a deterministic one that, whenever it has more than one option, investigates all possibilities. Because more than one option may be successful, this strategy entails that parser functions become multiple-valued. The usual step is to make the functions set-valued. This means that an expression like $f(g(\xi))$, where f and g are parser functions becomes something like

$$\{\eta | \zeta \in g(\xi) \wedge \eta \in f(\zeta)\}.$$

As a consequence, in the deterministic case the parser does not have the simple form it could have; its functions produce singletons instead of their elements.

Although one can try to hide the complexity introduced in the transition from nondeterministic functions to deterministic set-valued functions by using smart definitions (Hutton, 1993), or monads (Wadler, 1992), it would be preferable to avoid the problem altogether. In this paper we take a different view on multiple-valuedness to achieve this. A nondeterministic function can be seen as a deterministic one that returns a simple specification of a nondeterministic process, producing any of the results of the nondeterministic function, followed by the execution of the specified process. The task of the transformation to multiple-valuedness is to switch from executing the processes to interpreting their specifications. In this view, transforming nondeterministic functions to set-valued ones, amounts to proposing sets as specifications of processes. Of course, $\{e_1, e_2\}$ is a very ugly specification of a process that could produce either e_1 or e_2. The usual notation is $e_1|e_2$ (Hoare, 1985). A clear advantage of such expressions is that the specification of a deterministic process is just the value to be produced. Specifications like $e_1|e_2$ are called *bunches*, and the $|$ that separates the *elements* of the bunch is called the *bunch-union* operator. We refer to the property that a singleton bunch is exactly the same as its only element as the singleton property of bunches.

Functions distribute over nondeterministic values:

$$f(e_1|e_2) = f(e_1)|f(e_2).$$

The nondeterministic way to interpret this formula is: if f is applied to either e_1 or e_2 it produces either $f(e_1)$ or $f(e_2)$, which can both be nondeterministic. If we switch to interpreting specifications of processes, rather than executing them, the same formula means: if f is applied to the bunch $e_1|e_2$, the result is the bunch union of the bunch-valued results of $f(e_1)$ and $f(e_2)$. In other words, the requirement that bunches be interpreted implies that function application must distribute over bunch union. This distributivity property of bunches, together with their singleton property, make it possible to reinterpret the specification of a nondeterministic function as the specification of a deterministic bunch-valued one and vice versa, without transformations.

The distinction between sets and bunches resembles a quite common, but often implicitly made, distinction: the one between strings and lists. Strings have the singleton property, lists do not. We will advocate the use of both strings and bunches for defining the signatures of functions.

The paper starts with a discussion of bunches, strings and signatures. Bunch theory is subsequently illustrated by using it for the definition of top-down, Earley-like and LR(0) parsers. Bunch-valued functions can also be used to generalize conventional functional approaches as an alternative to going from a functional to a relational description. This will be shown by giving a bunch-functional account of a relational extension of attribute grammars that was proposed in (Hemerik, 1984), in the style of (Johnson, 1987). An earlier version of this paper appeared as (Leermakers, 1993b).

II. BUNCH THEORY

As we saw above, a bunch is a specification of a nondeterministic process. Alternatively, a bunch can be seen as a set-like aggregation, but it has some remarkable properties:

1. The singleton property: a singleton is the same thing as its only element; both specify a process that produces only that single element.

2. The process that corresponds to a bunch produces definite values. Therefore, the simplest specification of such a process is a simple enumeration of values. Thus, elements of a bunch cannot be bunches that are not definite values.

3. The distributivity property: function application distributes over bunch union.

Bunch union | has the same properties as set union: it is associative, commutative and idempotent. The main intuitive difference with sets is that a bunch is not 'one thing' if it has more than one element. This is why a bunch with many elements cannot be one element of another bunch: it can only be many elements of another bunch. This is also why a bunch with many elements cannot be passed to a function or operator as one thing. Here are a few examples of equalities for expressions involving bunches that illustrate the above:

$$3 + (1|2) \equiv 4|5$$
$$(3|4) + (1|2) \equiv 4|5|5|6 \equiv 4|5|6$$
$$cos(\pi|0) \equiv -1|1$$
$$(1|2) > 3 \equiv \mathbf{false}|\mathbf{false} \equiv \mathbf{false}$$

Apart from enumerations, one has the empty bunch *null*. It is the identity of bunch union. The bunch *all* denotes the bunch of all definite values. If e is one of the values in a bunch x, we write $e \leftarrow x$. Here and henceforth, e is a definite value or, what is the same, a singleton bunch.

Formalization

A formalization of bunches is the following:

- *null* is the empty bunch;
- if e is a definite value then e is a bunch;
- if x and y are bunches then $x|y$ is a bunch.

All bunch properties that do not follow directly from these clauses are captured by

- $(x|y)|z \equiv x|(y|z) \equiv x|y|z$;
- $x|x \equiv x$;
- $x|null \equiv x$;
- $f(x|y) \equiv f(x)|f(y)$,

where x, y, z are bunches and f is a function.

Bunch expressions

Bunch-valued functions are defined in terms of bunch expressions. A bunch expression *denotes* a bunch. A bunch expression is either a bunch, in which case it denotes itself, or it is some constructor taking other bunch expressions, in which case the bunch denoted is defined in terms of the bunches denoted by the constituent bunch expressions. A bunch expression may contain free variables, in which case the bunch denoted is indexed by the values of these variables. Unless stated otherwise, variables are definite; they cannot be bound to bunches with cardinality unequal to one. Below follows the list of constructors that will be used in this paper, with their meaning.

One constructor is bunch union. Hence, if x and y are bunch expressions then $x|y$ is a bunch expression. Not surprisingly, the denotation of $x|y$ is the bunch union of the bunches denoted by x and y.

The second constructor is if-then-else. Given a proposition P and bunch expressions x and y, the expression

$$\textbf{if } P \textbf{ then } x \textbf{ else } y \tag{1}$$

is a bunch expression. Normally, at least P will contain free variables. It will be clear that for each assignment of values to the variables the complex expression (1) is equivalent to the bunch denoted by x if P is true and to the bunch denoted by y otherwise.

Free variables in bunch expressions can be bound by λ-abstraction. If i is a variable and x is a bunch expression, then

$$\lambda i \cdot x$$

is a bunch-valued function. For any definite value e, the function application

$$\lambda i \cdot x(e)$$

is a bunch expression and denotes the bunch denoted by x, after substituting e for every occurrence of variable i in x. If f is a bunch-valued function and x a bunch expression, then $f(x)$ is a bunch expression denoting the result of distributing f over the bunch denoted by x. Note that it is important to distinguish between functions and expressions. In expression x in $\lambda i \cdot x$, variable i may occur more than once. If function $\lambda i \cdot x$ is applied to a bunch y, then the distributivity of functions over bunches means that the function applies to each $e \leftarrow y$ separately. That is, if $x = i + i$ then $\lambda i \cdot x(2|3) = (2+2)|(3+3)$; 2+3 is not included. In jargon, the semantics of bunch notation is such that functions are not *unfoldable*: a function invocation cannot necessarily be textually replaced by the expression that defines the function (Søndergard & Sestoft, 1990).

Our enterprise is to make it possible to reinterpret a nondeterministic functional definition as a multiple-valued one. Therefore, we shall finally define the ubiquitously used 'let...in...' construct in bunch terms. Following (Norvell & Hehner, 1992), we view the usual let construct as being composed of the guarded-expression construct

$$P \rhd x \overset{def}{\equiv} \textbf{if } P \textbf{ then } x \textbf{ else } \mathit{null},$$

where P is called a guard, and **let** *quantification*

$$\textbf{let } i \cdot x \overset{def}{\equiv} \lambda i \cdot x(\mathit{all}).$$

Guards may involve bunch expressions through conditions of the form $i \leftarrow x$, where x is a bunch expression and i a (definite) variable bound by **let** quantification. A simple example is

$$\textbf{let } i \cdot i \leftarrow x \; \rhd f(i),$$

which corresponds to 'let $i = x$ in $f(i)$' in a deterministic language, and to 'let i be *a* value of x in $f(i)$' in a nondeterministic language. To diminish the need for disambiguating brackets, we stipulate that $\rhd$ takes precedence over $|$ and **let**. The bunch *all* will in general be infinite, so that a function that distributes over it might produce an infinite bunch as well. In our application, however, the structure of bunch expressions will be such that **let**'s produce only finite bunches.

Laws

The following laws are useful for manipulating bunch expressions:

$$(P_1 \wedge P_2) \; \rhd x \equiv P_1 \; \rhd (P_2 \; \rhd x), \tag{2}$$

$$(P_1 \vee P_2) \; \rhd x \equiv (P_1 \; \rhd x)|(P_2 \; \rhd x), \tag{3}$$

$$P \equiv i \leftarrow \textbf{let } i' \cdot (P' \; \rhd i'), \tag{4}$$

$$\textbf{let } i \cdot i \leftarrow x \; \rhd f(i) \equiv f(x). \tag{5}$$

In (5), i is a variable that does not occur free in x. In (4), P' is obtained from P by replacing variable i by i'. Laws (2) to (5) are easy to prove: the first three follow from the definition of $\rhd$; the fourth follows from the distributivity of function application over bunches, and it is also quite obvious from the nondeterministic point of view.

Notation

A typical function definition looks like

$$f = \lambda X \cdot (\textbf{let} Y \cdot (\textbf{let} Z \cdot P(X,Y,Z) \; \rhd A(X,Y))) \tag{6}$$

where P is a predicate, A is a function and X, Y, Z are variables or strings of variables. In this paper, we will use a notational convention that removes the λ's and the **let**'s from definitions such as (6). Instead of (6) we write

$$f(X) = \quad P(X,Y,Z) \rhd A(X,Y). \tag{7}$$

So $\lambda X\cdot$ has changed into a formal argument on the left-hand side and we adopt the convention that free variables at the right-hand side (here Y,Z) are bound by **let**'s. The scope of such an implicit **let** is in practice always clear: it is from the first occurrence of the variable usually until "|", or else until the end of the bunch expression. Thus, whenever an expression $P \rhd x$ is encountered in this paper, with some free variables, its meaning is that all possible values for the free variables must be tried to make the guard P true and all results x must be combined in one bunch. Note that (7) is equivalent to

$$f(X) = \quad \exists_Z(P(X,Y,Z)) \rhd A(X,Y).$$

This equivalence may sometimes make algorithms easier to understand.

Strings

The distinction between sets and bunches may seem strange at first, but it resembles a quite common, but often implicitly made, distinction: the one between strings and lists (Hehner, 1993): strings have the singleton property, but lists do not.

In this paper, different kinds of strings are distinguished by their symbol for concatenation. The bunch $\Pi^*_\$$ of strings over the bunch Π, with concatenator \$, is defined as

- $\epsilon \leftarrow \Pi^*_\$$;
- if $e \leftarrow \Pi$ then $e \leftarrow \Pi^*_\$$;
- if $x \leftarrow \Pi^*_\$$ and $y \leftarrow \Pi^*_\$$ then $x\$y \leftarrow \Pi^*_\$$.

Variable \$ takes the value ',' (for argument strings, used as input and output of functions, see below) or it is empty, in which case elements are combined into strings by mere juxtaposition. In parsing theory, for instance, strings of grammar symbols are normally written without an explicit concatenator. We will follow this convention. String concatenation binds tighter than everything else, and concatenation by juxtaposition takes precedence over concatenation with ','.

Strings are subject to the equivalences

- $(x\$y)\$z = x\$(y\$z) = x\$y\$z$;
- $\epsilon\$x = x\$\epsilon = x$.

Hence, strings cannot be elements of strings of the same kind.

The formal definition of strings and the equivalences that hold for them are similar to those of bunches. Only, string concatenation is not commutative and not idempotent, and there is no distributivity law. Strings can be elements of bunches and of strings of a different kind (with another concatenator). String concatenation distributes over bunch union.

Function signature

Given functions $f : A \mapsto B$ and $g : B \mapsto C$, the signatures entail that the function fg defined by

$$(fg)(x) = g(f(x)) \tag{8}$$

is a type-correct composition that has signature $A \mapsto C$. If f is allowed to produce a bunch of elements of B, then $g(f(x))$ is well-formed as well, due to the distributivity property of bunches. This means that it is natural not to mark the bunch-valuedness of functions in their signatures. Thus, $f : A \mapsto B$ in general denotes a bunch-valued function. It can be combined with $g : B \mapsto C$ to produce a function $fg : A \mapsto C$, which will be multiple-valued if f or g is. Of course, it is not surprising that bunch-valuedness does not show up in signatures: nondeterminism also does not.

Usually, one writes function signatures like

$$f : A \times B \mapsto C \times D$$

where A, B, C, D are sets and $A \times B = \{(a, b) | a \in A \wedge b \in B\}$. If a function has more than one argument, we adopt the convention that it actually takes a string of arguments. To keep things symmetric, it also produces a string. Hence, we write

$$f : A, B \mapsto C, D$$

where A, B, C, D are bunches and A, B follows from the distributivity of ',' over A and B. The use of bunches as types thus eliminates the need for defining an operator on types for each operator on values (Hehner, 1993). The empty type, usually called *void*, is simply the empty string ϵ.

Summarizing, every function has one argument and one result, which are both strings (possibly singletons). Thus, a general function can be applied to a bunch of strings and produces a bunch of strings. See especially section VIII for advantages of this convention.

III. CONTEXT-FREE GRAMMARS: NOTATION

A context-free grammar is a four-tuple $G = (V_N, V_T, P, S)$, where S is the start symbol, V_N is the *bunch* of nonterminals, V_T is the bunch of terminals. Furthermore, $V = V_N | V_T$ is the bunch of grammar symbols. Relating to grammar symbols, the following typed variables are used: $x, y \leftarrow V_T$, $\xi, \eta, \rho, \zeta \leftarrow V_T^*$, $A, B \leftarrow V_N$, $X, Y \leftarrow V$, $\alpha, \beta, \gamma, \delta, \mu, \nu \leftarrow V^*$. Lastly, P is the collection of grammar rules. A grammar rule for nonterminal A, with right-hand side α, is denoted as $A \rightarrow \alpha$. If β can be derived from α in any number of steps, we write $\alpha \xrightarrow{*} \beta$.

IV. GENERAL RECURSIVE DESCENT PARSING

Given some input string ξ of terminal symbols, a grammar determines for each string of grammar symbols α whether or not ξ can be derived in any number of steps from α, i.e.

whether $\alpha \xrightarrow{*} \xi$. Also, for each substring η of ξ it may be determined whether or not $\alpha \xrightarrow{*} \eta$. Let us define for each α a bunch-valued *recognition function* $[\alpha] : V_T^* \mapsto V_T^*$, as follows:

$$[\alpha](\xi) = \quad \alpha \xrightarrow{*} \eta \wedge \xi = \eta\rho \ \rhd \rho. \tag{9}$$

Stated differently, this defines a function $[\cdot]$ that operates on two strings of grammar symbols, such that $[\cdot](\alpha, \xi) = [\alpha](\xi)$. Note that $\epsilon \leftarrow [S](\xi)$, equivalent to $S \xrightarrow{*} \xi$, means that ξ is a correct sentence.

In (9), the argument is split into two parts, the first of which is derivable from α. The second part is output by the function. It follows, for all α and β, that

$$\begin{aligned}
[\alpha\beta](\xi) = &\quad \alpha\beta \xrightarrow{*} \eta \wedge \xi = \eta\rho \ \rhd \rho \\
= &\quad \{\text{Split of } \alpha\beta \xrightarrow{*} \eta)\} \\
&\quad \alpha \xrightarrow{*} \eta_1 \wedge \xi = \eta_1\rho_1 \wedge \beta \xrightarrow{*} \eta_2 \wedge \rho_1 = \eta_2\rho \ \rhd \rho \\
= &\quad \{\text{Laws (2) and (4)}\} \\
&\quad \rho_1 \leftarrow (\alpha \xrightarrow{*} \eta_1 \wedge \xi = \eta_1\rho_1' \ \rhd \rho_1') \rhd (\beta \xrightarrow{*} \eta_2 \wedge \rho_1 = \eta_2\rho \ \rhd \rho) \\
= &\quad \{\text{Definition (9) of } [\alpha] \text{ and } [\beta]\} \\
&\quad \rho_1 \leftarrow [\alpha](\xi) \ \rhd [\beta](\rho_1) \\
= &\quad \{\text{Law (5)}\} \\
&\quad [\beta]([\alpha](\xi)).
\end{aligned}$$

Thus, $[\alpha\beta] = [\alpha][\beta]$, where $[\alpha][\beta]$ is the composition of functions $[\alpha]$ and $[\beta]$, defined by (8). In other words, $\alpha = X_1...X_k$ implies $[\alpha] = [X_1]...[X_k]$ and $[\epsilon](\xi) = \xi$. In algebraic terms, the mapping $[\cdot]$ is a homomorphism from V^* to a function space of bunch-valued functions. As the functions $[\alpha]$ are compositions of functions $[X]$, an implementation for the latter implies an implementation of the former. Now,

$$
\begin{aligned}
[X](\xi) = {} & X \overset{*}{\rightarrow} \eta \wedge \xi = \eta\rho \; \triangleright \rho \\
= {} & \{X \overset{*}{\rightarrow} \eta \text{ involves zero derivation steps, or at least one grammar rule is applied}\} \\
& ((X \leftarrow V_T \wedge X = \eta) \vee (X \rightarrow \beta \wedge \beta \overset{*}{\rightarrow} \eta)) \wedge \xi = \eta\rho \; \triangleright \rho \\
= {} & \{\text{Law (3)}\} \\
& (X \leftarrow V_T \wedge \xi = X\rho \; \triangleright \rho) \mid \\
& (X \rightarrow \beta \wedge \beta \overset{*}{\rightarrow} \eta \wedge \xi = \eta\rho \; \triangleright \rho) \\
= {} & \{\text{Law (2)}\} \\
& (X \leftarrow V_T \wedge \xi = X\rho \; \triangleright \rho) \mid \\
& (X \rightarrow \beta \; \triangleright (\beta \overset{*}{\rightarrow} \eta \wedge \xi = \eta\rho \; \triangleright \rho)) \\
= {} & \{\text{Definition (9)}\} \\
& (X \leftarrow V_T \wedge \xi = X\rho \; \triangleright \rho) \mid \\
& (X \rightarrow \beta \; \triangleright [\beta](\xi)).
\end{aligned}
$$

To summarize, we have, for terminals x and nonterminals A:

$$
\begin{aligned}
[x](\xi) = {} & \xi = x\rho \; \triangleright \rho, \\
[A](\xi) = {} & A \rightarrow \alpha \; \triangleright [\alpha](\xi), \\
[XY\beta](\xi) = {} & [Y\beta]([X](\xi)), \\
[\epsilon](\xi) = {} & \xi.
\end{aligned}
\tag{10}
$$

Note the use of the distributivity property of bunches in the third line. If rules have regular expressions at their right-hand sides, all this is easily extended (a, b denote regular expressions):

$$
\begin{aligned}
[x](\xi) = {} & \xi = x\rho \; \triangleright \rho, \\
[A](\xi) = {} & A \rightarrow a \; \triangleright [a](\xi), \\
[ab](\xi) = {} & [b]([a](\xi)), \\
[a|b](\xi) = {} & [a](\xi) \mid [b](\xi), && \text{(alternatives } a, b) \\
[\langle a \rangle](\xi) = {} & \xi \mid [a](\xi), && \text{(optional } a) \\
[\{a\}](\xi) = {} & \xi \mid [\{a\}]([a](\xi)), && \text{(iterative } a) \\
[\epsilon](\xi) = {} & \xi.
\end{aligned}
\tag{11}
$$

We use $\langle\rangle$ to denote optionality because the usual square brackets are used for recognition functions. The right-hand sides of lines three to six in (11) depend on a, b only via the functions $[a]$ and $[b]$. For this reason, these definitions are sometimes seen as applications of combinators, i.e., higher-order functions. With f, g denoting arbitrary bunch-valued functions from some domain (e.g., V_T^*) to itself, $\{f\}$, $[f]$, $f|g$ are other such functions, defined by

$$\begin{aligned}
(f|g)(\xi) &= \ f(\xi) \mid g(\xi), && \text{(alternatives } f, g\text{)} \\
\{f\}(\xi) &= \ \xi \mid \{f\}(f(\xi)), && \text{(iterative } f\text{)} \\
\langle f\rangle(\xi) &= \ \xi \mid f(\xi). && \text{(optional } f\text{)}
\end{aligned}$$

It follows that $[a|b] \equiv [a]|[b]$, $[\{a\}] \equiv \{[a]\}$, and $[\langle a\rangle] \equiv \langle[a]\rangle$. Finally, $[ab] \equiv [a][b]$, where $[a][b]$ is the functional composition of $[a]$ and $[b]$, defined in (8). In other words, the recognition function $[a]$ for regular expression a can be obtained by replacing every grammar symbol X that occurs in it by its function $[X]$ and interpreting all constructors in the regular expression (alternatives, concatenation, iteration, optionality) as combinators of recognition functions. For a detailed exposition of combinator parsing, see (Hutton, 1992).

The symbol '|' has been overloaded rather heavily by now. In some contexts, its meaning can even be ambiguous: $f|g$ can be an application of the combinator, or it can be a bunch of two functions. If confusion can arise, we distinguish with a subscript: $|_a$ is the alternatives operator of regular expressions, $|_b$ is bunch union, and $|_c$ is the combinator. The three can be combined in one line:

$$[a|_a b](\xi) = ([a]|_c[b])(\xi) = [a](\xi)|_b[b](\xi)$$

It was noted by (Meertens, 1986) that higher-order programming with nondeterministic functions may lead to paradoxes. This made (Norvell & Hehner, 1992) conclude that a nondeterministic function should be seen as a bunch of deterministic ones, as the ensuing distributivity property would prevent paradoxes. However, the decomposition of a nondeterministic function into a bunch of deterministic ones is far from trivial, and even not always possible. Moreover, there seems to be no fundamental reason why a nondeterministic function could not be seen as 'one thing'. Paradoxes only appear if the higher-order function uses a functional argument in more than one place in its body. The above combinators do not have this property, except for $\{f\}$. The 'paradox' in this case would be that $\{f|_c g\} \neq \{f\}|_c\{g\}$, which is quite logical in our context; the two sides are functions corresponding to different regular expressions. By contrast, one still has $\{f|_b g\} = \{f\}|_b\{g\}$: combinators distribute over bunches. Thus, there is no paradox, only confusion between $|_c$ and $|_b$.

Recursive-descent recognition functions like ours, with lists instead of bunches, were introduced in (Wadler, 1985) and reformulated as a parsing monad in (Wadler, 1990). A difference is that Wadler's functions produce lists of pairs that consist of a grammar symbol and unparsed input, whereas in our case functions produce only unparsed input. This difference is crucial in the monad context, because it means that our functions do not have monad structure: the type of $[X]$ does not depend on the type of X.

V. DETERMINISTIC RECURSIVE DESCENT PARSING

The singleton property of bunches is notationally convenient if one applies a general parsing technique to grammars for which the technique happens to provide a deterministic recognizer. If the general technique is defined with set-valued recognition functions, in the deterministic case all these functions produce sets with at most one value. If a function produces the empty set, this means that an error has been detected. If one works with bunch-valued functions instead, in a deterministic recognizer these produce *null* if an error has occurred and definite values otherwise.

There is a standard method to make parsing algorithms more deterministic: the addition of look-ahead (Aho & Ullman, 1977). The application of look-ahead techniques to recursive descent parsing involves two functions, *first* and *follow*:

$$first(\alpha) = \quad x \leftarrow V_T \wedge \alpha \overset{*}{\rightarrow} x\beta \rhd x,$$

$$\begin{aligned} follow(X) = \quad & A \rightarrow \alpha X \beta \ \rhd first(\beta) \mid \\ & A \rightarrow \alpha X \beta \wedge \beta \xrightarrow{*} \epsilon \ \rhd follow(A). \end{aligned}$$

Although *follow* not necessarily terminates if it is interpreted as an algorithm, it uniquely defines a smallest bunch $follow(X)$, for every X. It is convenient to add to each grammar the rule $S' \rightarrow S \perp$, where S' and $\perp$ are new symbols which appear only in this rule. S' is the new start symbol and $\perp$ is formally added to V_T. Of course, any correct input must now end with $\perp$. The above then implies that $\perp \leftarrow follow(S)$, and it is guaranteed that $follow(X) \neq null$ if $\exists_{\alpha\beta}(S \xrightarrow{*} \alpha X \beta)$. If X is one of the added symbols $S', \perp$ then $follow(X) = null$. It is not difficult to verify that if for $A \neq S'$ function $[A]$ is redefined as

$$[A](\xi) = \quad A \rightarrow \alpha \wedge \xi = x\eta \wedge (x \leftarrow first(\alpha) \vee (\alpha \xrightarrow{*} \epsilon \wedge x \leftarrow follow(A))) \ \rhd [\alpha](\xi),$$

the result of $[S'](\xi)$ is not affected. If for all $A \neq S'$ and every x at most one α exists that makes the guard true, the choice of grammar rule is always unique. This is the case for LL(1) grammars. For such grammars, the look-ahead technique makes each invocation $[A](\xi)$ produce either *null* if an error in the input string has been encountered, or a string of terminals that still have to be parsed; the general algorithm specializes to a deterministic recognizer.

VI. RECURSIVE ASCENT PARSING

Bunch notation is equally useful for bottom-up parsing. To illustrate this, let us start from the following specification of an Earley-like parser ($\delta \leftarrow (V_T^* | V_N V_T^*)$):

$$\begin{aligned} & [A \rightarrow \alpha \cdot \beta](\delta) = \\ & \qquad \delta \leftarrow V_T^* \wedge \beta \xrightarrow{*} \epsilon \ \rhd A\delta \mid \\ & \qquad \delta = X\eta\rho \wedge \beta \xrightarrow{*} X\eta \ \rhd A\rho. \end{aligned} \tag{12}$$

Note that the right-hand side does not depend on α, so that it would not be necessary to keep it at the left-hand side. Nevertheless, we use dotted rules ('items') $A \rightarrow \alpha \cdot \beta$ because they are the usual device to denote partially recognized grammar rules. The use of items comes down to an implicit rewriting of the grammar, in which the items are auxiliary nonterminals. Hence, $[A \rightarrow \alpha \cdot \beta]$ can be seen as a recognition function for nonterminal $A \rightarrow \alpha \cdot \beta$ of a cover for the original grammar (Leermakers, 1993a). Another motivation for keeping the α in $[A \rightarrow \alpha \cdot \beta]$ is that it helps to understand what the function does; see below. Moreover, the use of items simplifies the analogy of this section with section VII on LR-parsing, in which *states*, which are sets of items, play a similar role.

If applied to a string ξ of terminal symbols, specification (12) reduces to

$$[A \rightarrow \alpha \cdot \beta](\xi) = \quad \beta \xrightarrow{*} \eta \wedge \xi = \eta\rho \ \triangleright A\rho. \tag{13}$$

This means that, after adding a rule $S' \rightarrow S$ to the original grammar, it follows that

$$S' \leftarrow [S' \rightarrow \cdot S](\xi)$$

if and only if ξ is a correct sentence.

The intuition behind (12) is the following. The input sentence being $x_1 ... x_n$, function invocation $[A \rightarrow \alpha \cdot \beta](\delta)$ occurs only if there are i, j $(0 \leq i \leq j \leq n)$ such that $\alpha \xrightarrow{*} x_{i+1} ... x_j$, and either $j = n$ and δ is empty or $\delta = X\rho$ and there is a k $(j \leq k \leq n)$ such that $X \xrightarrow{*} x_{j+1} ... x_k$ and $\rho = x_{k+1} ... x_n$. Function invocation $[A \rightarrow \alpha \cdot \beta](\delta)$ investigates which prefixes of δ can be derived from β. The prefix may be not be empty if δ starts with a nonterminal. Each prefix of δ derives $x_{j+1} .... x_l$ for some l $(j \leq l \leq n)$. If the prefix can be derived from β, β derives this same input string and A derives $x_{i+1} .... x_l$, the concatenation of the strings derived by α and β. The function thus returns A, followed by $x_{l+1} ... x_n$, the part of the input sentence that has not yet been parsed. If more than one prefix can be found, all answers are returned as a bunch. The result of the function can be seen as a rewritten version of the string $\alpha\delta$; it removes a prefix from $\alpha\delta$ that includes at least α and replaces it by a nonterminal that derives the prefix: $\alpha\delta$ is rewritten in a bottom-up way.

We strive for an implementation of (12) of the recursive ascent type. To this end, we note that $\beta \xrightarrow{*} X\eta$ means that either X is introduced by a grammar rule $B \rightarrow \mu X \nu$, with $\mu \xrightarrow{*} \epsilon$, or X is already in β: $\beta = \mu X \nu$, with $\mu \xrightarrow{*} \epsilon$:

$[A \to \alpha \cdot \beta](\delta) =$

$\quad \delta \leftarrow V_T^* \wedge \beta \xrightarrow{*} \epsilon \rhd A\delta \mid$

$\quad \delta = X\eta\rho \wedge \beta = \mu X \nu \wedge \mu \xrightarrow{*} \epsilon \wedge \nu \xrightarrow{*} \eta \rhd A\rho \mid$

$\quad \delta = X\eta_2\eta_1\rho \wedge \beta \xrightarrow{*} B\eta_1 \wedge B \to \mu X \nu \wedge \mu \xrightarrow{*} \epsilon \wedge \nu \xrightarrow{*} \eta_2 \rhd A\rho$

$=$ {Use (2) in the second and third line, adding an extra clause $\beta \xrightarrow{*} B\gamma$ (this is optional; it is to make the result more efficient, see below) and introducing auxiliary variable ζ}

$\quad \delta \leftarrow V_T^* \wedge \beta \xrightarrow{*} \epsilon \rhd A\delta \mid$

$\quad \delta = X\zeta \wedge \beta = \mu X \nu \wedge \mu \xrightarrow{*} \epsilon \rhd (\nu \xrightarrow{*} \eta \wedge \zeta = \eta\rho \rhd A\rho) \mid$

$\quad \delta = X\eta_2\eta_1\rho \wedge \beta \xrightarrow{*} B\gamma \wedge B \to \mu X \nu \wedge \mu \xrightarrow{*} \epsilon \wedge \nu \xrightarrow{*} \eta_2 \rhd (\beta \xrightarrow{*} B\eta_1 \rhd A\rho)$

$=$ {Apply (13) in the second line (applicable because $\zeta \leftarrow V_T^*$) and (12) in the third}

$\quad \delta \leftarrow V_T^* \wedge \beta \xrightarrow{*} \epsilon \rhd A\delta \mid$

$\quad \delta = X\zeta \wedge \beta = \mu X \nu \wedge \mu \xrightarrow{*} \epsilon \rhd [A \to \alpha\mu X \cdot \nu](\zeta) \mid$

$\quad \delta = X\eta_2\eta_1\rho \wedge \beta \xrightarrow{*} B\gamma \wedge B \to \mu X \nu \wedge \mu \xrightarrow{*} \epsilon \wedge \nu \xrightarrow{*} \eta_2 \rhd [A \to \alpha \cdot \beta](B\eta_1\rho)$

$=$ {Change $\eta_1\rho$ into ρ, η_2 into η, and use (4) (i is $B\rho$) and (13) ($\eta\rho$ consists of terminals) in the third line}

$\quad \delta \leftarrow V_T^* \wedge \beta \xrightarrow{*} \epsilon \rhd A\delta \mid$

$\quad \delta = X\zeta \wedge \beta = \mu X \nu \wedge \mu \xrightarrow{*} \epsilon \rhd [A \to \alpha\mu X \cdot \nu](\zeta) \mid$

$\quad \delta = X\eta\rho \wedge \beta \xrightarrow{*} B\gamma \wedge B \to \mu X \nu \wedge \mu \xrightarrow{*} \epsilon \wedge B\rho \leftarrow [B \to \mu X \cdot \nu](\eta\rho) \rhd [A \to \alpha \cdot \beta](B\rho).$

Using (5) and changing $\eta\rho$ into ζ, one finally obtains

$$\begin{aligned}
&[A \to \alpha \cdot \beta](\delta) = \\
&\quad \delta \leftarrow V_T^* \wedge \beta \xrightarrow{*} \epsilon \rhd A\delta \mid \\
&\quad \delta = X\zeta \wedge \beta = \mu X \nu \wedge \mu \xrightarrow{*} \epsilon \rhd [A \to \alpha\mu X \cdot \nu](\zeta) \mid \\
&\quad \delta = X\zeta \wedge \beta \xrightarrow{*} B\gamma \wedge B \to \mu X \nu \wedge \mu \xrightarrow{*} \epsilon \rhd [A \to \alpha \cdot \beta]([B \to \mu X \cdot \nu](\zeta)).
\end{aligned} \tag{14}$$

The conciseness of the last line is due to the distributivity property of bunches. In deriving (14) a critical need is that not $B \leftarrow V_T$, in other words, that terminals and nonterminals are disjoint.

Note that if a function $[A \to \alpha \cdot \beta]$ is invoked by another function, then its argument δ is in V_T^*. It may recursively call itself with rewritten versions of this δ, i.e., with prefixes of δ replaced by some nonterminal B, until this B appears in β in such a way that the symbols before B (in β) may derive the empty string.

The recognizer terminates for all non-cyclic grammars. Note that the conditions

$$\begin{aligned}
&\beta \xrightarrow{*} \epsilon \\
&\beta = \mu X \nu \wedge \mu \xrightarrow{*} \epsilon \\
&\exists_\gamma(\beta \xrightarrow{*} B\gamma) \wedge B \rightarrow \mu X \nu \wedge \mu \xrightarrow{*} \epsilon
\end{aligned}$$

are independent of the input string, and for every β, X the values of μ, ν, B that make them true can be computed before parsing. To get an efficient implementation such precomputation is to be compounded with function memoization (Leermakers, 1992). It is simple to add loop detection to the memoization technique to get an algorithm that always terminates (Leermakers, 1993a).

In the case of a grammar without ϵ-rules, (14) becomes even simpler:

$$\begin{aligned}
&[A \rightarrow \alpha \cdot \beta](\delta) = \\
&\qquad \delta \leftarrow V_T^* \wedge \beta = \epsilon \rhd A\delta \mid \\
&\qquad \delta = X\zeta \wedge \beta = X\nu \rhd [A \rightarrow \alpha X \cdot \nu](\zeta) \mid \\
&\qquad \delta = X\zeta \wedge \beta \xrightarrow{*} B\gamma \wedge B \rightarrow X\nu \rhd [A \rightarrow \alpha \cdot \beta]([B \rightarrow X \cdot \nu](\zeta)).
\end{aligned}$$

As far as I know, the recognizer of this section is a new variant of Earley-like parsing. In (Leermakers, 1992) a closely related algorithm was given, with two functions per dotted rule, instead of one. The functional parsing algorithm given in (Matsumoto et al., 1983) is also quite similar to ours, even though it does not involve dotted rules.

Two aspects of (14) contribute to its simplicity compared to conventional formulations of related algorithms: the way bunch notation handles 'nondeterminism' and the fact that memoization (which corresponds to using a parse matrix) is separated from the core algorithm. For a discussion of the relation of the above algorithm with the standard Earley parser, see (Leermakers, 1993a).

VII. LR(0) PARSING

Let us now give a recursive ascent LR(0) parser. The derivation is analogous to the derivation of (14). In fact, recursive-ascent LR(0) parsing appears if one tries to make recursive ascent Earley-like parsing more deterministic by combining items into *states*, i.e., sets of items. Because of the analogy with the preceding section, we just give the specification and the implementation of the recognizer, and skip the proof.

The specification of the recognition function is, with q a state:

$$\begin{aligned}
&[q](\delta) = \\
&\quad \delta \leftarrow V_T^* \wedge A \rightarrow \alpha \cdot \beta \in q \wedge \beta \xrightarrow{*} \epsilon \rhd A \rightarrow \alpha \cdot \beta, \delta \mid \\
&\quad \delta = X\zeta \wedge A \rightarrow \alpha \cdot \beta \in q \wedge \beta \xrightarrow{*} X\eta \wedge \zeta = \eta\rho \rhd A \rightarrow \alpha \cdot \beta, \rho.
\end{aligned} \tag{15}$$

To formulate the implementation, we need the auxiliary functions *ini*, defined as

$$ini(q) = \{B \rightarrow \cdot\nu | B \rightarrow \nu \wedge A \rightarrow \alpha \cdot \beta \in q \wedge \beta \xrightarrow{*} B\gamma\},$$

and the function *goto*:

$$goto(q, X) = \{A \rightarrow \alpha X \cdot \beta | A \rightarrow \alpha \cdot X\beta \in (q \cup ini(q))\}.$$

Then the following is an implementation of (15):

$$\begin{aligned}
&[q](\delta) = \\
&\quad \delta \leftarrow V_T^* \wedge A \rightarrow \alpha \cdot \beta \in q \wedge \beta \xrightarrow{*} \epsilon \rhd A \rightarrow \alpha \cdot \beta, \delta \mid \\
&\quad \delta = X\zeta \wedge A \rightarrow \alpha \cdot \mu X\gamma \in q \wedge \mu \xrightarrow{*} \epsilon \wedge \\
&\qquad A \rightarrow \alpha\mu X \cdot \gamma, \rho \leftarrow [goto(q, X)](\zeta) \rhd A \rightarrow \alpha \cdot \mu X\gamma, \rho \mid \\
&\quad \delta = X\zeta \wedge C \rightarrow \cdot\mu X\nu \in ini(q) \wedge \mu \xrightarrow{*} \epsilon \wedge \\
&\qquad C \rightarrow \mu X \cdot \nu, \rho \leftarrow [goto(q, X)](\zeta) \rhd [q](C\rho).
\end{aligned} \tag{16}$$

This algorithm is functionally equivalent to an LR(0) parser if the grammar has no ϵ-rules, though it does not use a parse stack. That is, if the grammar is LR(0), (16) is a deterministic function. It consists of one function $[\cdot]$ with two arguments (q and δ), or, in the alternative view, of as many single-argument functions $[q]$ as there are states q, just like the recursive-ascent LR parsers of (Kruseman Aretz, 1988) and (Barnard & Cordy, 1988), which were the first to totally avoid the use of parse stacks.

For some left-recursive grammars with ϵ-rules, conventional nondeterministic LR(0) parsers do not terminate (Tomita, 1986). The above algorithm terminates as long as the grammar is not cyclic, and can be quite efficient if recognition functions are memoized. Termination for any grammar can be achieved by adding loop detection to the memoization mechanism.

VIII. ATTRIBUTE GRAMMARS

Bunches can also be used in the theory of attribute grammars. In conventional attribute grammars, each attribute has an associated function that computes its value in terms of the values of other attributes. It is very natural to take such an attribute function to be bunch-valued. If the function produces *null*, this means that the computation of its attribute fails. If it produces a bunch with more than one element, attribute computation is ambiguous. Bunch-valued attribute functions are particularly apt for natural-language parsing, since both failure and ambiguity of attribute computation are natural phenomena in this application of attribute grammars.

An attribute grammar is based on a context-free grammar, called the *backbone* grammar. Each grammar symbol X of this grammar has a number of attributes. Its attributes come in two kinds: inherited ones, the type of which is denoted collectively as $inh(X)$, and synthesized ones, with type $syn(X)$.

It is convenient to extend the mappings *inh* and *syn* to V^*, as follows:

$$inh(\epsilon) = \epsilon \text{ (`void')}$$

$$inh(\alpha\beta) = inh(\alpha), inh(\beta)$$

and the same for *syn*:

$$syn(\epsilon) = \epsilon$$
$$syn(\alpha\beta) = syn(\alpha), syn(\beta)$$

Note that $inh(\alpha), inh(\beta)$ is the concatenation of two bunches of strings. If types would have been sets then $inh(\alpha) \times inh(\beta)$ would have been a set of pairs, which is quite different!

Below we will use bunch-valued attribute functions f_α that have domain $inh(\alpha)$ and range $syn(\alpha)$:

$$f_\alpha : F_\alpha, \text{ where } F_\alpha = inh(\alpha) \mapsto syn(\alpha).$$

A function of type F_α can be concatenated with a function of type F_β. The result is a function of type $F_{\alpha\beta}$ (inh_α,syn_α are variables of the types $inh(\alpha)$,$syn(\beta)$, respectively):

$$\begin{aligned} & f_\alpha \# f_\beta = f_{\alpha\beta}, \text{ where} \\ & f_{\alpha\beta}(inh_\alpha, inh_\beta) = \quad syn_\alpha \leftarrow f_\alpha(inh_\alpha) \wedge syn_\beta \leftarrow f_\beta(inh_\beta) \;\triangleright\; syn_\alpha, syn_\beta \end{aligned}$$

There is one unique function f_ϵ such that $f_\epsilon \# f_\alpha = f_\alpha \# f_\epsilon = f_\alpha$.

Formally, an attribute grammar can be defined to consist of

1. A context-free backbone grammar G;
2. A function f_x of type F_x for each terminal symbol x;
3. A function $f^{A\to\alpha}$ for each grammar rule $A \to \alpha$, of type $inh(A \to \alpha) \mapsto syn(A \to \alpha)$, where

$$inh(A \to \alpha) = inh(A), syn(\alpha)$$
$$syn(A \to \alpha) = syn(A), inh(\alpha)$$

To define formally how attribute functions are associated with grammar symbols, we define *dressed derivations.* A dressed direct derivation is defined by

$$\alpha A\beta, f_\alpha \# f_A \# f_\beta \;\to\; \alpha\gamma\beta, f_\alpha \# f_\gamma \# f_\beta \;\overset{def}{\equiv}\; A \to \gamma \wedge f_A = f_\gamma \circ f^{A\to\gamma}, \tag{17}$$

where $\circ$ is defined by

$$f_\alpha \circ f^{A\to\alpha} = f_A, \text{ where}$$
$$f_A(inh_A) = \quad syn_A, inh_\alpha \leftarrow f^{A\to\alpha}(inh_A, syn_\alpha) \wedge syn_\alpha \leftarrow f_\alpha(inh_\alpha) \rhd syn_A.$$

This definition is formally circular: syn_α depends on inh_α, which in turn depends on syn_α. If $inh(\alpha), syn(\alpha)$ are structured domains, however, not every aspect of syn_α necessarily depends on every aspect of inh_α, so that the function produced by a $\circ$ composition may well be effective (Johnson, 1987).

The transitive and reflexive closure of dressed derivations is defined just as for normal derivations. Then, to each derivation $S \overset{*}{\to} x_1...x_n$ corresponds a dressed derivation

$$S, f_S \;\overset{*}{\to}\; x_1...x_n, f_{x_1}\#...\#f_{x_n}.$$

Parsing algorithms for attribute grammars with bunch-valued attribute functions are identical to algorithms for conventional attribute grammars. For example, the above recursive ascent parser can be modified such that it becomes a parser for attribute grammars:

$$\begin{aligned}
&[A \to \alpha\cdot\beta](\delta, f_\delta, f_\alpha) = \\
&\quad \delta \leftarrow V_T^* \wedge (\beta, f_\beta \overset{*}{\to} \epsilon, f_\epsilon) \rhd A\delta, (f_\alpha\#f_\beta)\circ f^{A\to\alpha\beta})\#f_\delta \mid \\
&\quad \delta = X\zeta \wedge f_\delta = f_X\#f_\zeta \wedge \beta = \mu X\nu \wedge (\mu, f_\mu \overset{*}{\to} \epsilon, f_\epsilon) \rhd [A \to \alpha\mu X\cdot\nu](\zeta, f_\zeta, f_\alpha\#f_\mu\#f_X) \mid \\
&\quad \delta = X\zeta \wedge f_\delta = f_X\#f_\zeta \wedge \beta \overset{*}{\to} B\gamma \wedge B \to \mu X\nu \wedge (\mu, f_\mu \overset{*}{\to} \epsilon, f_\epsilon) \\
&\qquad\qquad\qquad\qquad \rhd[A \to \alpha\cdot\beta]([B \to \mu X\cdot\nu](\zeta, f_\zeta, f_\mu\#f_X), f_\alpha).
\end{aligned}$$

Note the way in which the three arguments of $[A \to \alpha\cdot\beta]$ in the last line are assembled as the concatenation of the result of the call of $[B \to \mu X\cdot\nu]$ and f_α.

After the recognition, one may execute $f_{S'}()$ (which has no arguments assuming that S' has no inherited attributes) to produce the required attributes, i.e., the synthesized attributes of S'. Hence, one has the following parser:

$$parser(\xi) = \quad (S', f_{S'}) \leftarrow [S' \to \cdot S](\xi, f_\xi, f_\epsilon) \rhd f_{S'}(),$$

and $f_\xi = f_{x_1}\#...\#f_{x_n}$ if $\xi = x_1...x_n$ is the input sentence.

IX. CONCLUSIONS

Parsing theory should be formulated in a functional or relational way. In any case, imperative formulations and push-down automata belong to the realm of implementation details. It is a mystery why simple, stack-less, formulations of LR parsing were found as late as 1988. The dominance of imperative programming and automata theory must have played a role in the delay. To the degree that the deterministic case is considered to have a special status, a bunch-functional approach is to be preferred over a relational one, because one is interested in relations that are almost functions. If the theory of a parser is developed functionally, this does not mean that parsers must be implemented in a functional programming language, nor that the implementation must be recursive; it is not forbidden to implement recursive algorithms with an explicit recursion stack to suit particular needs.

This paper is meant to establish, by way of illustrative examples, that the bunch concept is a mathematical notion as respectable as sets and lists. The reader is invited to translate any of the sections into set notation and observe the notational burden that he/she has to add. The conciseness of bunch notation is not accidental: one writes a multiple-valued algorithm by writing its nondeterministic counterpart, without adding a 'process interpretation' part. In addition to being shorter, algorithms defined in bunch notation are often more readable and more intuitive.

One could argue that almost the same conciseness can be obtained using sets and extra operators to distribute functions over sets and to switch between singletons and their elements. However, one should keep in mind that the bunch notion is more primitive than its set relative: a bunch is an aggregation, a set is an encapsulated aggregation (Hehner, 1993). It is the encapsulation aspect of sets that leads to conceptual problems, to students (a set that contains nothing is not nothing) as well as to scientists (the set that contains everything, including itself, leads to a paradox). Being essentially simpler, bunches are not troubled by such intricacies. In practice, it is fine to implement bunches with sets, as long as one keeps in mind the difference between a notion and its implementation. After all, the possibility of implementing sets in terms of lists does not mean that sets can be dispensed with. One distinguishing aspect of bunch-valued functions, which goes beyond notational issues, is that normal functions are embedded in them.

The notion of bunches has been introduced in (Hehner, 1984). Sets with nondeterministic interpretation, like bunches, were also proposed in (Hughes & O'Donnell, 1990). In (Wadler, 1992) a kind of bunch-valued lambda-calculus is discussed. Bunch-valued functions also appear in (Meertens, 1986), (Bauer et al., 1987) and (Norvell & Hehner, 1992), as nondeterministic specifications of programs.

I refer to (Hehner, 1993) for further elaborations on the bunch theme, and many other applications. In (Leermakers, 1993a) bunch notation is adopted as a tool for the formulation of parsing theory, in the spirit of this paper.

Acknowledgement

I thank (Lex Augusteijn | Paul Jansen | Frans Kruseman Aretz | Mark-Jan Nederhof | Theo Norvell | a referee) for his useful comments.

REFERENCES

Aho A.V. & Ullman J.D. (1977) *Principles of Compiler Design* (Addison-Wesley).

Barnard D.T. & Cordy J.R. (1988) SL parses the LR languages, *Computer Languages* 13(2), 65-74.

Bauer F.L., Ehler H., Horsch A., Möller B., Partsch H., Puakner O. & Pepper P. (1987) *The Munich Project CIP: Volume II: The Program Transformation System CIP-S*, Lecture Notes in Computer Science 292 (Springer-Verlag).

Hehner E.C.R. (1984) *The Logic of Programming* (Prentice-Hall).

Hehner E.C.R. (1993) *a Practical Theory of Programming* (Springer-Verlag).

Hemerik C. (1993) *Formal Definitions of Programming Languages as a Basis for Compiler Construction*, Ph.D. Thesis, Technical university, Eindhoven, The Netherlands.

Hoare C.A.R. (1985) *Communicating Sequential Processes* (Prentice-Hall).

Hughes J. & O'Donnell J. (1990), Expressing and reasoning about non-deterministic functional programs, *Functional Programming*, (Glasgow 1989), edited by K. Davis and J. Hughes, Workshops in Computing (Springer-Verlag).

Hutton G. (1992) Higher-order functions for parsing, *Journal of Functional Programming* 2(3), 323-343.

Johnson T. (1993) *Conference on Functional Programming Languages and Computer Architecture*, LNCS 274 (1987), 154-173.

Kruseman Aretz F.E.J. (1988) On a recursive ascent parser, *Information Processing Letters* 29, 201-206.

Leermakers R. (1992) A recursive ascent Earley parser, *Information Processing Letters* 41, 87-91.

Leermakers R. (1993a) *The Functional Treatment of Parsing* (Kluwer Academic Publishers).

Leermakers R. (1993b) The use of Bunch Notation in Parsing Theory, *Proceedings of the Third International Workshop on Parsing Technologies* (Tilburg & Durbuy), 135-144.

Matsumoto Y., Tanaka H., Hirakawa H., Miyoshi H. & Yasukawa H. (1983) BUP: a bottom-up parser embedded in Prolog, *New Generation Computing* 1(2).

Norvell T.S. & Hehner E.C.R. (1992) Logical Specifications for Functional Programs, *Proceedings of the Second International Conference on the Mathematics of Program Construction* (Oxford).

Søndergard & Sestoft (1990) Referential Transparency, Definiteness and Unfoldability, *Acta Informatica* 27, 505-517.

Tomita M. (1986) *Efficient Parsing for Natural Language, A Fast Algorithm for Practical Systems* (Kluwer Academic Publishers, Boston).

Wadler P. (1985) How to replace failure by a list of successes, *Conference on Functional Programming Languages and Computer Architecture* (Nancy, France); LNCS 201 (Springer-Verlag), 113-128.

Wadler P. (1990) Comprehending Monads, *Conference on Functional Programming Languages and Computer Architecture.*

Wadler P. (1992) The essence of functional programming, *19th Annual Symposium on Principles of Programming Languages* (Santa Fe).