The Use of Bunch Notation in Parsing 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 parsing theory 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 a formal semantics for context-free grammars and of functional parsing algorithms.

1 Introduction

The semantics of a context-free grammar can be given in a number of ways. In the three most important interpretations, a grammar is viewed as a rewriting system, or as a set of inequalities, or as an abstract program. The latter two interpretations are discussed in this paper, using a variant of set notation, called *bunch notation*. Subsequently, a new Earley-like recursive-ascent parser is formulated with the same notation.

There are two major differences between sets and bunches. One difference is that a bunch with one element is identified with that one element (the singleton property). Moreover, a function or operator that is defined on some domain may be applied to a bunch of elements that belong to that domain. Such an application, say $f(X)$, causes the function f to be applied to each separate element of the bunch X, after which the results are combined in a bunch, which is the result of $f(X)$. This is called the distributivity property of bunches. To define bunch-valued expressions, a variant of the notation of (Norvell — Hehner, 1992) is used.

A language can be defined as a bunch of strings rather than as a set of strings. Then, both the singleton and distributivity properties of bunches simplify the formalization of the natural interpretation of context-free grammars, in which grammar rules are seen as constraints on the possible assignments of languages to nonterminals.

Multiple-valued (recursive descent and ascent) parsing functions can be defined as bunch-valued functions. Again, both the singleton and distributivity properties have advantages. The singleton property smoothes the transition from parsing algorithms for general grammars to deterministic algorithms for LL(k) and LALR(k) grammars. The distributivity property makes it possible to write succinct formulae in bunch notation, that 'blow up' if translated into set notation. Finally, Norvell — Hehner's (1992) bunch notation has an advantage over standard set notation when it comes to defining functional algorithms, in that it resembles traditional notation for specifying programs. In particular, the definition of recursive parsing algorithms is in terms of a construct akin to Dijkstra's guarded commands (Dijkstra, 1976).

The paper starts with an introduction to bunch notation. The first application of the notation is a reformulation of the natural semantics of context-free grammars. Subsequent sections give functional definitions of known recursive descent algorithms and a new recursive ascent recognition algorithm.

2 Bunch notation

In standard mathematics, a (total) function $f : \mathcal{A} \mapsto \mathcal{B}$ associates one element of $\mathcal{B}$ with each element of $\mathcal{A}$. A function is a special case of a relation, which may associate any number of elements of $\mathcal{B}$ with each element of $\mathcal{A}$. Conversely, each relation can be seen as a special kind of a function too: a set-valued function that, if applied to $a \in \mathcal{A}$, yields the set of elements of $\mathcal{B}$ associated with a by the relation. Alternatively, a relation may be viewed as a nondeterministic function: of all elements of $\mathcal{B}$ related to some element $a \in \mathcal{A}$, the nondeterministic function arbitrarily picks one.

The set-valuedness of functions associated with a relation has one peculiar consequence. Take a function $f : \mathcal{A} \mapsto \mathcal{B}$, and view it as a relation with the special property that it relates only one element of $\mathcal{B}$ to each element of $\mathcal{A}$. Next, use the mapping from relations to set-valued or nondeterministic functions to view the relation as a function again. Then one would expect to re-obtain the original function f. If the relation is mapped to a nondeterministic function, this is indeed the case: the function happens to be deterministic and is equal to f. Using standard sets, however, the set-valued function associated with the relation associated with f, produces a set with exactly one element (a singleton) where f produces that element. This suggests that it is better not to see relations as set-valued functions, but rather as *bunch-valued* functions. A bunch is a set with some non-standard properties, so that it can be interpreted in an alternative way: a bunch is also a process that nondeterministically produces one of its values. The alternative interpretation implies the following three properties of bunches:

1. The process that corresponds to a bunch with one value (a singleton) is deterministic: it can only produce that one value. Therefore: a singleton is identified with its only element.

2. The process that corresponds to a bunch produces definite values. Therefore: elements of bunches can not be bunches with cardinality $\neq 1$.

3. If f is a function and x is a bunch that can produce the values $e_1 ... e_k$, then $f(x)$ can take the values $f(e_1) ... f(e_k)$. Therefore: functions distribute over bunches.

With these properties, a bunch simultaneously allows two interpretations. In the set-valued interpretation it is just a collection of values. In the nondeterministic interpretation one value is randomly taken out of this collection.

Bunches are the result of bunch expressions. Given two bunch expressions x and y, their bunch union $x|y$ denotes a process that could either produce a value of x or a value of y. Bunch union has the same properties as set union: it is associative, commutative and idempotent. The main 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 bunch expressions that illustrate the above:

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

If e is one of the values a bunch expression x can take, we write $e \leftarrow x$. Here and henceforth, e is a definite value or, what is the same, a singleton bunch. As definite values are also bunches, and elements of bunches cannot be bunches with cardinality unequal to one, the distinction between $\in$ and $\subseteq$ is no longer needed: if, for all e, $e \leftarrow x$ implies $e \leftarrow y$ then we write $x \leftarrow y$. In words, x is a sub-bunch of y, or, x is smaller than y.

Bunch expressions can be simple or complex. The simplest simple bunch expression is the empty bunch *null*. It is the identity of bunch union. Other simple bunch expressions are enumerations. The following is a formal definition of simple bunch expressions with elements from a (possibly infinite) set of definite values:

1. *null* is a simple bunch expression;

2. if e is a definite value then e is a simple bunch expression;

3. if x and y are simple bunch expressions then $x|y$ is a simple bunch expression.

A simple bunch expression may be rewritten into an equivalent simple bunch expression using $(x|y)|z \equiv x|(y|z) \equiv x|y|z$, $x|y \equiv y|x$, $x|x \equiv x$ and $x|null \equiv x$. Assuming some ordering on the set of definite values, it is not difficult to define a canonical form for each simple bunch expression, which may serve as a unique representation of the bunch denoted. The bunch expression *all* denotes the smallest bunch such that $e \leftarrow all$ for all definite values e.

Complex bunch expressions are constructed with variables. Unless stated otherwise, variables have types that consist of definite values only. Such variables are called definite; they cannot be bound to bunches with cardinality unequal to one. Given a proposition P and bunch expressions x and y, the expression

$$\textbf{if } P \textbf{ then } x \textbf{ else } y \qquad (1)$$

is a complex bunch expression. It contains free variables if P, x, or y contain free variables (nontrivial P do). It will be clear that for each assignment of values to the variables the complex expression (1) is equivalent to x if P is true and to 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,

$$\lambda i \cdot x(e) \stackrel{def}{\equiv} \text{Substitute } e \text{ for free variables } i \text{ everywhere in } x.$$

This definition holds only for *definite* values e. Note, therefore, 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 $x = mult(i, i)$ then $\lambda i \cdot x(2|3) = mult(2,2)|mult(3,3)$; $mult(2,3)$ is not included. In jargon, our functional language is characterized as having a semantics such that functions are not *unfoldable*: a function invocation cannot be textually replaced by the expression that defines the function, if function parameters are not definite (Sondergard — Sesoft, 1990).

This is practically all we need to know about bunches. Let us just add some notations:

$$P \rhd x \stackrel{def}{\equiv} \textbf{if } P \textbf{ then } x \textbf{ else } null$$

$$\textbf{let } i \cdot x \stackrel{def}{\equiv} \lambda i \cdot x(all)$$

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.

The following laws are useful for manipulating bunch expressions:

$$(P_1 \wedge P_2) \rhd x \equiv P_1 \rhd (P_2 \rhd x), \qquad (2)$$

$$(P_1 \vee P_2) \rhd x \equiv (P_1 \rhd x)|(P_2 \rhd x), \qquad (3)$$

$$\textbf{let } i \cdot (i \leftarrow x \rhd f(i)) \equiv f(x). \qquad (4)$$

In (4) it is assumed that i does not occur free in x. These laws are easy to prove: the first two follow from the definition of $\rhd$, and the third follows from the distributivity of function application over bunches.

Normally, set-valued functions are defined using set comprehension according to the schema

$$f(X) = \{A(X,Y) | \exists_Z P(X,Y,Z)\}, \qquad (5)$$

where P is a predicate, A is a function and X, Y, Z are variables or sets of variables. Now let us define a related bunch-valued function, called f_b:

$$f_b = \lambda X \cdot (\textbf{let} Y \cdot (\textbf{let} Z \cdot (P(X,Y,Z) \rhd A(X,Y)))) \qquad (6)$$

It follows that f and f_b are equivalent if the latter is interpreted as producing a set. A nice aspect of (6) is that its algorithmic content is more explicit than the algorithmic content of (5); because **let** is defined as a function application to *all*, it is explicit that (6) involves searching over all values of Y, Z.

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

$$f_b(X) = \quad P(X,Y,Z) \rhd A(X,Y). \qquad (7)$$

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 of the free variables must be tried to make the guard P true and all results x must be combined in one bunch.

Lastly, note that (7) is equivalent to

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

That is, both functions produce the same bunch for every X. The algorithmic interpretation of both formulae is not exactly the same, however (see below). Therefore, if a variable appears only in a guard, like Z in (7), it is implicitly subject to existential quantification.

Algorithmic interpretation

In what follows, bunch-valued functions are either known computable functions, or their definitions have the following general format:

$$\begin{aligned} f(X) = \; & P_1(X,Y_1) \rhd A_1(X,Y_1) \mid \\ & P_2(X,Y_2) \rhd A_2(X,Y_2) \mid \\ & \vdots \\ & P_k(X,Y_k) \rhd A_k(X,Y_k), \end{aligned}$$

where X is a collection of input parameters and Y_i are collections of variables subject to **let** quantification. P_i are predicates and A_i are bunch-valued functions. Both P_i and A_i may involve other applications of bunch-valued functions. The intention is that bunches are interpreted as collections: bunch-valued functions produce all their results simultaneously. Function f then has a simple algorithmic (imperative) interpretation, which makes use of a bunch-valued variable "result":

$f(X) =$ result:=*null*;
for all Y_1 such that $P_1(X,Y_1)$ do
result:=result | $A_1(X,Y_1)$
od;
for all Y_2 such that $P_2(X,Y_2)$ do
result:=result | $A_2(X,Y_2)$
od;
.
.
.
for all Y_k such that $P_k(X,Y_k)$ do
result:=result | $A_k(X,Y_k)$
od;
return result

The invocations $A_i(X,Y_i)$ and function applications inside P_i are to be computed in the same vein. In this algorithmic interpretation, a function may or may not terminate. If it does not terminate, $f(X)$ does not define an algorithm. This may happen if the definition of $f(X)$ is circular, i.e. if the computation of some P_i or A_i involves $f(X)$ again.

The above function f is *deterministic* if for each X at most one proposition $P_i(X,Y_i)$ can be true, for only one value of Y_i, and function A_i is deterministic.

3 The natural semantics of grammars

Within the family of rewriting systems, context-free grammars have a distinguishing property: they have a declarative meaning. This means that a grammar can be understood not only by producing a sample of trial sentences with it, but also by viewing it as a collection of static statements about the language to be defined. This is the underlying reason for their intelligibility and their usefulness. In the natural interpretation, grammar symbols are seen as variables over languages and grammar rules as stipulations of relations between these variables. A grammar, in this view, is analogous to a collection of arithmetic inequalities with variables. Take, for instance, the following inequalities:

$$k \geq l + 3, \; l \geq 5.$$

A formal interpretation of this is that there are two symbols k and l here, that there is some assignment function h from these symbols to numbers, and that the inequalities restrict the possible values of h, via

$$h(k) \geq h(l) + 3,\ h(l) \geq 5.$$

Of course, there are still many functions h that satisfy these constraints but there is one that assigns the smallest possible numbers to the symbols: $h(k) = 8, h(l) = 5$.

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 $\alpha \xrightarrow{*} \beta$.

Languages

A language is a bunch of strings of terminals, i.e. a subbunch of V_T^*. Concatenation is an operation that is defined for (pairs of) strings. Therefore, it distributes over languages L and M, if these are concatenated:

$$LM = \ \xi \leftarrow L \wedge \rho \leftarrow M \ \triangleright \xi\rho. \qquad (8)$$

This equation is referred to as the definition of *language multiplication*, although it is not really a definition: it follows from the distributivity property.

The interpretation

A nonterminal can be seen as a variable of type language (like k, l are variables of type integer), a terminal is a constant language (like 3,5 are constant integers). Just like in the arithmetic example, we assume an assignment function that performs the mapping from symbols to their interpretation. This function is called L_G, as its value will be determined by the grammar G. Take, for example, the grammar rule $A \rightarrow xBy$. In the natural interpretation this rule means

$$xL_G(B)y \leftarrow L_G(A).$$

That is, the grammar rule is a constraint on L_G. In principle, L_G need only apply to nonterminals, but it is convenient to extend it, via

$$L_G(x) = x, \text{ for all } x \leftarrow V_T,$$

to all grammar symbols. Moreover, we further extend it to arbitrary strings of grammar symbols, via

$$\begin{aligned} L_G(\alpha\beta) &= L_G(\alpha)L_G(\beta), \\ L_G(\epsilon) &= \epsilon, \end{aligned} \qquad (9)$$

so that the following equalities hold true:

$$xL_G(B)y = L_G(x)L_G(B)L_G(y) = L_G(xBy).$$

Equation (9) states that L_G not only maps grammar-symbol strings into languages: it also maps an operation on its input objects (concatenation) to an operation on its output objects (language multiplication). In other words, the extended L_G is a homomorphism.

The interpretation of any grammar rule $A \rightarrow \alpha$ now reads

$$L_G(\alpha) \leftarrow L_G(A).$$

In other words, the language associated with A and the language associated with α are related: the latter is a sub-bunch of the former. Just like in the arithmetic example, a collection of such inequalities does not define the assignment function uniquely. There is one L_G, however, that assigns the smallest possible languages to grammar symbols. This smallest homomorphism is what the grammar is intended to define.

Inspired by the arithmetic analogue one may write X instead of $L_G(X)$ and insist that $A \rightarrow \alpha$ means $\alpha \leftarrow A$: α is a sub-bunch of A. Rules $A \rightarrow \alpha_1$, ...,$A \rightarrow \alpha_k$, for the same nonterminal, are often abbreviated to $A \rightarrow \alpha_1|...|\alpha_k$. This is very natural here; it means that $\alpha_1|...|\alpha_k$ is a subbunch of A.

A rule $A \rightarrow \alpha_1|...|\alpha_k$ involves a list of alternative strings. For completeness, and for later reference, let us give the formal semantics of more general rules $A \rightarrow a$, with a denoting arbitrary regular expressions. As above, this semantics is $L_G(a) \leftarrow L_G(A)$, where the application of L_G to

regular expressions is defined by (a, b are regular expressions)

$$\begin{aligned} L_G(ab) &= L_G(a)L_G(b), && \text{(concatenation)}\\ L_G(a|b) &= L_G(a) \mid L_G(b), && \text{(alternation)}\\ L_G(\langle a\rangle) &= \epsilon \mid L_G(a), && \text{(optionality)}\\ L_G(\{a\}) &= \epsilon \mid L_G(\{a\})L_G(a), && \text{(iteration)} \end{aligned} \tag{10}$$

where the brackets $\langle\ \rangle$ were used for optionality instead of the more usual [ ] to avoid confusion later on.

4 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* $[\alpha]$ from V_T^* to V_T^*, as follows:

$$[\alpha](\xi) = \alpha \xrightarrow{*} \eta \wedge \xi = \eta\rho \rhd \rho. \tag{11}$$

Stated differently, this defines a function $[\cdot]$ that operates on two strings of grammar symbols, such that $[\cdot](\alpha, \xi) = [\alpha](\xi)$. Similar recognition functions, with lists instead of bunches, were introduced in (Wadler, 1985). Note that $\epsilon \leftarrow [S](\xi)$, equivalent to $S \xrightarrow{*} \xi$, means that ξ is a correct sentence.

In (11), 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

$$\begin{aligned} [\alpha\beta](\xi) &= \alpha\beta \xrightarrow{*} \eta \wedge \xi = \eta\rho \rhd \rho\\ &= \alpha \xrightarrow{*} \eta_1 \wedge \xi = \eta_1\rho_1 \wedge\\ &\quad\ \beta \xrightarrow{*} \eta_2 \wedge \rho_1 = \eta_2\rho \rhd \rho\\ &= \alpha \xrightarrow{*} \eta_1 \wedge \xi = \eta_1\rho_1 \rhd\\ &\quad\ (\beta \xrightarrow{*} \eta_2 \wedge \rho_1 = \eta_2\rho \rhd \rho)\\ &= \rho_1 \leftarrow [\alpha](\xi) \rhd [\beta](\rho_1)\\ &= [\beta]([\alpha](\xi)). \end{aligned}$$

Here (2) and (4) were used in the second and fourth transitions, respectively. Thus, $[\alpha\beta] = [\alpha][\beta]$, where $[\alpha][\beta]$ is the composition of functions $[\alpha]$ and $[\beta]$, defined by

$$(fg)(\xi) = g(f(\xi)). \tag{12}$$

In other words, $\alpha = X_1 \ldots X_k$ implies $[\alpha] = [X_1]\ldots[X_k]$ 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,

$$\begin{aligned} [X](\xi) &= X \xrightarrow{*} \eta \wedge \xi = \eta\rho \rhd \rho\\ &= ((X \leftarrow V_T \wedge X = \eta) \vee\\ &\quad\ (X \rightarrow \beta \wedge \beta \xrightarrow{*} \eta)) \wedge \xi = \eta\rho \rhd \rho\\ &= (X \leftarrow V_T \wedge \xi = X\rho \rhd \rho)|\\ &\quad\ (X \rightarrow \beta \wedge \beta \xrightarrow{*} \eta \wedge \xi = \eta\rho \rhd \rho)\\ &= (X \leftarrow V_T \wedge \xi = X\rho \rhd \rho)|\\ &\quad\ (X \rightarrow \beta \rhd (\beta \xrightarrow{*} \eta \wedge \xi = \eta\rho \rhd \rho))\\ &= (X \leftarrow V_T \wedge \xi = X\rho \rhd \rho)|\\ &\quad\ (X \rightarrow \beta \rhd [\beta](\xi)). \end{aligned}$$

Here (3) was used to eliminate the disjunction $\vee$ and (2) to eliminate a conjunction $\wedge$. To summarize, we have, for terminals x and nonterminals A:

$$\begin{aligned} [x](\xi) &= \xi = x\rho \rhd \rho,\\ [A](\xi) &= A \rightarrow \alpha \rhd [\alpha](\xi),\\ [XY\beta](\xi) &= [Y\beta]([X](\xi)),\\ [\epsilon](\xi) &= \xi. \end{aligned} \tag{13}$$

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 (compare with (10)):

$$\begin{aligned} [x](\xi) &= \xi = x\rho \rhd \rho,\\ [A](\xi) &= A \rightarrow a \rhd [a](\xi),\\ [ab](\xi) &= [b]([a](\xi)),\\ [a|b](\xi) &= [a](\xi) \mid [b](\xi),\\ [\langle a\rangle](\xi) &= \xi \mid [a](\xi),\\ [\{a\}](\xi) &= \xi \mid [\{a\}]([a](\xi)),\\ [\epsilon](\xi) &= \xi. \end{aligned} \tag{14}$$

The right-hand sides of lines three to six in (14) 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

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

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 (12). 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 (alternation, concatenation, iteration, optionality) as combinators of recognition functions. For a detailed exposition of combinator parsing, see (Hutton, 1992). (Norvell — Hehner, 1992) issued a warning that higher-order programming with bunch-valued functions may lead to paradoxes that were noted by (Meertens, 1986) in the case of nondeterministic functions. The above combinators do not suffer from problems of this kind.

5 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*:

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

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

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 fully natural deterministic recognizer.

6 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 $(\delta \leftarrow (V_T^* | V_N V_T^*))$:

$$[A \rightarrow \alpha \cdot \beta](\delta) = \\ \delta \leftarrow V_T^* \wedge \beta \xrightarrow{*} \epsilon \rhd A\delta \mid \\ \delta = X\zeta \wedge \beta \xrightarrow{*} X\eta \wedge \zeta = \eta\rho \rhd A\rho. \qquad (15)$$

If applied to a string ξ of terminal symbols, this specification reduces to

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

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 this is that a function invocation $[A \rightarrow \alpha \cdot \beta](\delta)$ investigates which prefixes of δ can be rewritten to β, in a bottom-up way (using grammar rules from right to left). If a non-empty prefix can be found, this corresponds to a part of the input sentence, which is a string rewritable to the first symbol of δ (i.e. X) followed by the remainder of the prefix (which are terminals). If

$\beta \xrightarrow{*} \epsilon$, the prefix may be empty. Assuming that the function is invoked only if a directly preceding part of the input sentence was rewritable to α, it is deduced that this preceding part, followed by the part that corresponds to the found prefix of δ, can be rewritten into A. The function thus returns A, followed by the part of the input sentence that has not yet been parsed. If more than one prefix can be found, the function delivers a bunch.

We strive for an implementation of (15) 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 $\mu \xrightarrow{*} \epsilon$:

$$
\begin{aligned}
&[A \rightarrow \alpha \cdot \beta](\delta) = \\
&\quad \delta \leftarrow V_T^* \wedge \beta \xrightarrow{*} \epsilon \rhd A\delta \mid \\
&\quad \delta = X\zeta \wedge \beta = \mu X\nu \wedge \mu \xrightarrow{*} \epsilon \wedge \nu \xrightarrow{*} \eta \wedge \\
&\qquad \zeta = \eta\rho \rhd A\rho \mid \\
&\quad \delta = X\zeta \wedge \beta \xrightarrow{*} B\eta_1 \wedge B \rightarrow \mu X\nu \wedge \\
&\qquad \mu \xrightarrow{*} \epsilon \wedge \nu \xrightarrow{*} \eta_2 \wedge \zeta = \eta_2\eta_1\rho \rhd A\rho.
\end{aligned}
$$

After a few elementary rewriting steps using (15), one finally obtains

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

The conciseness of the last line is due to the distributivity property of bunches. In deriving (16) a critical need is that not $B \leftarrow V_T$, in other words, that terminals and nonterminals are disjoint.

Note that if a function $[A \rightarrow \alpha \cdot \beta]$ 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 pre-computation is to be compounded with function memoization (Leermakers, 1992; 1993).

In the case of a grammar without ϵ-rules, (16) becomes even simpler:

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

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. For a discussion of the relation of the above algorithm with the standard Earley parser, see (Leermakers, 1993). An analogous LR parser, with one function for each state (and, of course, without a parse stack), is also constructed in (Leermakers, 1993).

7 Conclusions

This paper should serve two purposes. Firstly, it should show the beauty of functional parsing theory. Secondly, the 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.

One could argue that almost the same conciseness can be obtained using normal sets and an extra ('map') operator to distribute functions over sets. 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 conciseness of bunch notation is not its only virtue. Functions defined with bunch notation look more 'algorithmic' than their translation into set notation, which is not unimportant if one wants to define an algorithm, if only for pedagogical reasons.

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; Norvell — Hehner, 1992), as nondeterministic specifications of programs.

I refer to (Hehner, 1993) for further elaborations on the bunch theme, and many other applications. This work also proposes to make a distinction between strings and sequences, which also exists between bunches and sets: strings have the singleton property, but sequences do not. As is apparent from the notation for elements of V^*, it is natural to make no distinction between grammar symbols and elements of V^* that have length one. Thus, elements of V^* are strings, not sequences. In (Leermakers, 1993) bunch notation is adopted as a tool for the formulation of parsing theory, in the spirit of this paper. In this book, bunches are also 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.

Acknowledgement

I thank Theo Norvell for his useful comments on the first draft of this paper, and Lex Augusteijn, Paul Jansen, Frans Kruseman Aretz and Mark-Jan Nederhoffor their constructive remarks about the second draft. Triggered by (Norvell — Hehner, 1992), it was Lex Augusteijn who inspired me to use bunches for the kind of parsing algorithms we are both engaged in.

References

Aho A.V. — J.D. Ullman (1977) *Principles of Compiler Design.* Reading, MA: Addison-Wesley.

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

Dijkstra E.W. (1976) *A Discipline of Programming.* London: Prentice Hall.

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

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

Hughes J. — J. O'Donnell (1990), "Expressing and reasoning about non-deterministic functional programs". In: K. Davis and J. Hughes (Eds), *Functional Programming.* Berlin: Springer-Verlag.

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

Leermakers R. (1992) "A recursive ascent Earley parser". In: *Information Processing Letters* 41, 87–91.

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

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

Norvell T.S. — E.C.R. Hehner (1992) "Logical Specifications for Functional Programs". In: *Proceedings of the Second International Conference on the Mathematics of Program Construction.* Oxford: Oxford University Press.

Sondergard — Sesoft (1990) "Referential Transparency, Definiteness and Unfoldability". *Acta Informatica* 27, 505–517.

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

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