Theoretical Computer Science 104 (1992) 299–312
Elsevier

Note

Recursive ascent parsing: from Earley to Marcus

René Leermakers

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

Communicated by M. Nivat
Received January 1991
Revised October 1991

Abstract

Leermakers, R., Recursive ascent parsing: from Earley to Marcus, Theoretical Computer Science 104 (1992) 299–312.

An overview is given of recursive ascent parsing, a new functional implementation technique for parsers for context-free grammars. The theory behind it unifies the treatment of hitherto virtually unrelated parsing methods, such as the Earley algorithm and LR parsing. This is effected by banning the distinguishing factors, such as stacks and parsing tables: stacks are replaced by recursive functions, parsing tables by function memoization. In addition to this unification, the theory provides a high-level view on parsing compared to the standard theory. Nevertheless, the functional implementations are as efficient as conventional ones, especially if the functions that constitute the parsers are formulated in low-level imperative languages with efficient function calls, like C. This means that they are important in practice, for parsing both artificial and natural languages. The recursive ascent treatment of LR(0) parsing has a natural generalization that leads to a new family of look-ahead parsers. This family is akin to the one proposed by Marcus a decade ago to model the processing of a natural language in the human mind.

1. Introduction

Recently, the theory of LR parsing gained new impetus by the discovery of the recursive ascent implementation technique for deterministic [1, 4, 11, 12] and non-deterministic [5, 7] LR parsers. In short, the novelty is that LR parsers can be implemented purely functionally and that this implementation has very simple correctness proofs. In its primary form, a recursive ascent parser consists of two functions for each state. In this paper we present the recursive ascent implementation for

a number of parsers, in ascending order of complexity. The simplest one is the Earley parser, after which the LR(0) parser is derived in analogy. The LR(0) parser is the simplest element of a class of look-ahead parsers that we refer to as Marcus parsers, and a study of this class closes the paper. Recursive ascent parsing is akin to recursive descent parsing and the paper starts with the presentation of a variant of recursive descent parsing, to give the reader a chance to get used to our notations.

We will present only the recognition part of parsers, i.e. we do not discuss the creation of parse trees, or whatever kind of trace of the parsing process. Especially for ambiguous grammars, the compact representation of the set of parse trees is a real issue, which we address in another paper [7].

2. Recursive Descent

Consider CF grammar $G=(V_N, V_T, P, S)$, with terminals V_T and nonterminals V_N. Let $V=V_N \cup V_T$. A well-known top-down parsing technique is the recursive descent parser. Recursive descent parsers consist of a number of procedures, usually one for each nonterminal. Here we present a variant that consists of functions, one for each so-called item (dotted rule). Items, grammar rules with a dot somewhere in the right-hand side, are used ubiquitously in parsing theory to denote a partially recognized rule: if $A \to \alpha\beta$ is a grammar rule (with Greek letters for arbitrary elements of V^*), then $A \to \alpha.\beta$ is an item. We overload the symbol $\to$, as usual, to let it also denote the *derives* relation:

$$\alpha \to \beta \equiv \exists_{\delta\gamma B\mu}(\alpha=\delta B\gamma \wedge \beta=\delta\mu\gamma \wedge B\to\mu\in P),$$

which implies that $B\to\mu$ is synonymous to $B\to\mu\in P$.

We use the operator $[\cdot]$ to map each item to its function:

$$[A\to\alpha.\beta]: N\mapsto 2^N,$$

where N is the set of integers, or a subset $0\ldots n_{\max}$, with $n_{\max}$ the maximum sentence length, and 2^N is the power set of N. The functions are to meet the following specification:

$$[A\to\alpha.\beta](i)=\{\, j \mid \beta \overset{*}{\to} x_{i+1}\ldots x_j\},$$

with $x_1\ldots x_n$ as the string to be parsed. So, the function reports which parts of the string can be derived from β starting from position i. A more constructive definition of the same functions follows from discerning three cases:

(1) Suppose $\beta=\varepsilon$. Then $\beta \overset{*}{\to} x_{i+1}\ldots x_j$ is equivalent to $i=j$ and, hence,

$$[A\to\alpha.](i)=\{i\}.$$

(2) Suppose $\beta = b\gamma$, with $b \in V_T$. Then

$$[A \to \alpha.b\gamma](i) = \{ j \mid b = x_{i+1} \wedge \gamma \xrightarrow{*} x_{i+2} \ldots x_j \}.$$

Now, if $A \to \alpha.b\gamma$ is an item then also $A \to \alpha b.\gamma$ is one and

$$\gamma \xrightarrow{*} x_{i+2} \ldots x_j \equiv j \in [A \to \alpha b.\gamma](i+1).$$

Hence,

$$[A \to \alpha.b\gamma](i) = \{ j \mid b = x_{i+1} \wedge j \in [A \to \alpha b.\gamma](i+1) \}.$$

(3) Suppose $\beta = B\gamma$, with $B \in V_N$. Then

$$[A \to \alpha.B\gamma](i) = \{ j \mid \exists_{k\delta} \colon B \to \delta \wedge \delta \xrightarrow{*} x_{i+1} \ldots x_k \wedge \gamma \xrightarrow{*} x_{k+1} \ldots x_j \}.$$

Replacing $B \to \delta \wedge \delta \xrightarrow{*} x_{i+1} \ldots x_k$ by $k \in [B \to .\delta](i)$ and $\gamma \xrightarrow{*} x_{k+1} \ldots x_j$ by $j \in [A \to \alpha B.\gamma](k)$, we conclude that

$$[A \to \alpha.B\gamma](i) = \{ j \mid k \in [B \to .\delta](i) \wedge j \in [A \to \alpha B.\gamma](k) \}.$$

In the last equation we left out the quantification over k, δ, and from now on we will systematically neglect the existential quantifications in definitions of this kind. All in all, we have a recursive definition of $[A \to \alpha.\beta](i)$ for each possible β, and these definitions together form a functional algorithm. If we add a grammar rule $S' \to S$ to G, with $S' \notin V$, then $S \xrightarrow{*} x_1 \ldots x_n$ is equivalent to $n \in [S' \to .S](0)$, so that the algorithm is to be invoked by calling $[S' \to .S](0)$. The algorithm works for any CF grammar except for grammars for which $\exists_{A\alpha\beta}(A \to \alpha \wedge \alpha \xrightarrow{*} A\beta)$. For such left-recursive grammars the recognizer does not terminate, as execution of $[A \to .\alpha](i)$ leads to a call of $[A \to .\alpha](i)$. The recognition is not a linear process in general: the function calls $[A \to \alpha.B\gamma](i)$ lead to calls $[B \to .\delta](i)$ for all values of δ such that $B \to \delta$ is a grammar rule.

3. Recursive ascent Earley

To be able to construct an implementation of $[A \to \alpha.\beta]$ that has fewer problems with left-recursive grammars, we need the so-called predict sets. Let $predict(A \to \alpha.\beta)$ be the set of initial items, that are derived from $A \to \alpha.\beta$ by the closure operation:

$$predict(A \to \alpha.\beta) = \{ B \to .\mu \mid B \to \mu \wedge \beta \Rightarrow^* B\gamma \}.$$

The double arrow $\Rightarrow$ denotes a *leftmost symbol* rewriting with a non-ε grammar rule, i.e.

$$\alpha \Rightarrow \beta \equiv \exists_{B\gamma\delta}(\alpha = B\gamma \wedge \beta = \delta\gamma \wedge B \to \delta \wedge \delta \neq \varepsilon).$$

A recursive ascent recognizer may be obtained by relating to each item $A \to \alpha.\beta$ not only the above $[A \to \alpha.\beta]$, but also a function that we take to be the result of applying the operator $\overline{[\cdot]}$ to the item:

$$\overline{[A \to \alpha.\beta]} : V \times N \mapsto 2^N.$$

It has the specification ($X \in V$)

$$\overline{[A \to \alpha.\beta]}(X, i) = \{ j \mid \beta \overset{*}{\Rightarrow} X\gamma \wedge \gamma \overset{*}{\to} x_{i+1} \ldots x_j \}.$$

Assuming x_{n+1} to be some end of sentence marker that is not in V, it can never be that $\beta \overset{*}{\Rightarrow} x_{n+1}\gamma$; hence, $\overline{[A \to \alpha.\beta]}(x_{n+1}, n+1) = \emptyset$. For $i \leqslant n$ the above functions are recursively implemented by (Algorithm 1)

$$\begin{aligned}
[A \to \alpha.\beta](i) = {} & \overline{[A \to \alpha.\beta]}(x_{i+1}, i+1) \\
& \cup \{ j \mid B \to .\varepsilon \in predict(A \to \alpha.\beta) \wedge j \in \overline{[A \to \alpha.\beta]}(B, i) \} \\
& \cup \{ i \mid \beta = \varepsilon \}, \qquad (1) \\
\overline{[A \to \alpha.\beta]}(X, i) = {} & \{ j \mid \beta = X\gamma \wedge j \in [A \to \alpha X.\gamma](i) \} \\
& \cup \{ j \mid j \in \overline{[A \to \alpha.\beta]}(C, k) \wedge C \to .X\delta \in predict(A \to \alpha.\beta) \wedge \\
& \qquad k \in [C \to X.\delta](i) \}
\end{aligned}$$

Proof. First we note that

$$\begin{aligned}
\beta \overset{*}{\to} x_{i+1} \ldots x_j \equiv {} & \exists_\gamma (\beta \overset{*}{\Rightarrow} x_{i+1}\gamma \wedge \gamma \overset{*}{\to} x_{i+2} \ldots x_j) \\
& \vee \exists_{B\gamma} (\beta \overset{*}{\Rightarrow} B\gamma \wedge B \to \varepsilon \wedge \gamma \overset{*}{\to} x_{i+1} \ldots x_j) \\
& \vee (\beta = \varepsilon \wedge i = j).
\end{aligned}$$

Substituting this in the specification of $[A \to \alpha.\beta]$ one gets

$$\begin{aligned}
[A \to \alpha.\beta](i) = {} & \{ j \mid \beta \overset{*}{\Rightarrow} x_{i+1}\gamma \wedge \gamma \overset{*}{\to} x_{i+2} \ldots x_j \} \\
& \cup \{ j \mid B \to \varepsilon \wedge \beta \overset{*}{\Rightarrow} B\gamma \wedge \gamma \overset{*}{\to} x_{i+1} \ldots x_j \} \\
& \cup \{ j \mid \beta = \varepsilon \wedge i = j \}.
\end{aligned}$$

This directly leads to the implementation given above because

$$A \to \alpha\beta \wedge \exists_\gamma (\beta \overset{*}{\Rightarrow} X\gamma \wedge \gamma \overset{*}{\to} x_{i+1} \ldots x_j) \equiv j \in \overline{[A \to \alpha.\beta]}(X, i),$$

$$A \to \alpha\beta \wedge \exists_\gamma (\beta \overset{*}{\Rightarrow} B\gamma) \wedge B \to \varepsilon \equiv B \to .\varepsilon \in predict(A \to \alpha.\beta).$$

For establishing the correctness of $\overline{[A\to\alpha.\beta]}$ note that $\beta \overset{*}{\Rightarrow} X\gamma$ either contains zero steps, in which case $\beta = X\gamma$, or contains at least one step:

$$\exists_\gamma(\beta \overset{*}{\Rightarrow} X\gamma \wedge \gamma \overset{*}{\to} x_{i+1}\ldots x_j) \equiv \exists_\gamma(\beta = X\gamma \wedge \gamma \overset{*}{\to} x_{i+1}\ldots x_j)$$

$$\vee \exists_{C\delta\gamma k}(\beta \overset{*}{\Rightarrow} C\gamma \wedge C\to X\delta \wedge \delta \overset{*}{\to} x_{i+1}\ldots x_k$$

$$\wedge \gamma \overset{*}{\to} x_{k+1}\ldots x_j).$$

Hence, $\overline{[A\to\alpha.\beta]}(X, i)$ may be written as the union of two sets, S_0 and S_1:

$$S_0 = \{j \mid \beta = X\gamma \wedge \gamma \overset{*}{\to} x_{i+1}\ldots x_j\},$$

$$S_1 = \{j \mid \beta \overset{*}{\Rightarrow} C\gamma \wedge C\to X\delta \wedge \delta \overset{*}{\to} x_{i+1}\ldots x_k \wedge \gamma \overset{*}{\to} x_{k+1}\ldots x_j\}.$$

With the specification of $[A\to\alpha X.\gamma]$, S_0 may be rewritten as

$$S_0 = \{j \mid \beta = X\gamma \wedge j \in [A\to\alpha X.\gamma](i)\}.$$

The set S_1 may be rewritten using the specifications of $\overline{[A\to\alpha.\beta]}(C, k)$ and *predict*$(A\to\alpha.\beta)$:

$$S_1 = \{j \mid j \in \overline{[A\to\alpha.\beta]}(C, k) \wedge C\to.X\delta \in predict(A\to\alpha.\beta) \wedge \delta \overset{*}{\to} x_{i+1}\ldots x_k\}.$$

With the definition of $[C\to X.\delta]$ one finally gets

$$S_1 = \{j \mid j \in \overline{[A\to\alpha.\beta]}(C, k) \wedge C\to.X\delta \in predict(A\to\alpha.\beta) \wedge k \in [C\to X.\delta](i)\}.$$

□

3.1. Complexity

The above recognizer needs exponential-time resources unless the functions are implemented as memo-functions. Memo-functions memorize for which arguments they have been called. If a function is called with the same arguments as before, the function returns the previous result without recomputing it. In conventional programming languages memo-functions are not available, but they can easily be implemented. The use of memo-functions obsoletes the introduction of devices like parse matrices [2]. The worst-case complexity analysis of the memoized recognizer is quite simple. Let n be the sentence length, $|G|$ the number of items of the grammar, p the maximum number of different left-hand sides in a predict set, and q the maximum number of items in a predict set with the same symbol after the dot. Then, there are $O(|G|pn)$ different invocations of recognizer functions. Each invocation of a function $\overline{[I]}$ invokes $O(qn)$ other functions, that all result in a set with $O(n)$ elements. The merge of these sets into one set with no duplicates can be accomplished in $O(qn^2)$ time on

a random-access machine. Hence, the total time complexity is $O(|G|pqn^3)$. The space needed for storing function results is $O(n)$ per invocation, i.e. $O(|G|pn^2)$ for the whole recognizer. These complexity results are almost identical to the usual ones for Earley parsing. Only the dependence on the grammar variables $|G|$, p and q differs slightly. Worst-case complexities need not be relevant in practice. We claim that for many practical grammars the present algorithm is more efficient than the existing implementations, for the following reason. The above formulae can be interpreted as defining two functions, $[\cdot]$ and $\overline{[\cdot]}$, that will work for variable grammars and strings. This view is convenient when building prototypes. If efficiency is an issue, however, one should precompute as much as possible and actually create, for a fixed grammar, the functions $[I]$ and $\overline{[I]}$ for every item I. In the terminology of functional programming the functions $[\cdot]$ and $\overline{[\cdot]}$ are to be evaluated partially for each item. In this way, the grammar is compiled into a collection of functions, just like conventional parser generators compile a grammar into LR tables or a recursive descent parser. Quite some work that is done at parse time by the standard Earley parser, such as the creation of *predict* sets and the processing of item sets, is transferred to compile time when transforming the grammar into a functional parser. As a consequence, the compiled parser is more efficient than the standard implementations of the Earley parser.

The above considerations only hold if our algorithm terminates. If the grammar has a cyclic derivation $A \rightarrow^+ A$, the execution of $\overline{[I]}(A, i)$ leads to a call of itself, and the algorithm does not terminate. Also, there may be a cycle of transitions labeled by nonterminals that derive ε. This occurs if for some k one has that, for $i = 1 \ldots k$,

$$A_{i+1} \rightarrow .\alpha_{i+1}\beta_{i+1} \in predict(A_i \rightarrow \alpha_i.\beta_i) \wedge \alpha_i \rightarrow^+ \varepsilon,$$

while

$$A_1 = A_{k+1} \wedge \alpha_1 = \alpha_{k+1} \wedge \beta_1 = \beta_{k+1}.$$

Then the execution of $[A_1 \rightarrow \alpha_1.\beta_1](i)$ leads to a call of itself, and the algorithm does not terminate. A cycle of this form occurs *iff* there is a derivation $A \rightarrow^+ \alpha A\beta$ such that $\alpha \rightarrow^+ \varepsilon$. It is easy, however, to define a variant of the recognizer that has no problems with these derivations. It is obtained from dropping the restriction that the leftmost symbol derivation $\Rightarrow$ may not use ε-rules; see [6].

4. Recursive ascent LR(0)

We now know how to cope with the problem of left-recursion. It is possible to change things slightly as to also avoid some unnecessary non-determinism, and this leads to LR-parsing.

The mechanism for reducing nondeterminism is the merging of functions corresponding to a number of competing items into one function. Let the set of all items of

G be given by I_G. Subsets of I_G are called states, and we use q to denote an arbitrary state. We associate with each state q a function, re-using the above operator $[\cdot]$,

$$[q]: N \mapsto 2^{I_G \times N}$$

that meets the specification

$$[q](i) = \{(A \to \alpha.\beta, j) \mid A \to \alpha.\beta \in q \wedge \beta \overset{*}{\to} x_{i+1} \ldots x_j\}.$$

As above, the function reports which parts of the sentence can be derived. But as the function is associated with a set q of items, it has to do so for each item in q. If we define the initial state $q_0 = \{S' \to .S\}$, we have that $S \overset{*}{\to} x_1 \ldots x_n$ is equivalent to $(S' \to .S, n) \in [q_0](0)$.

To be able to construct a recursive ascent implementation of $[q]$ we again need some auxiliary functions, similar to the *predict* function before. Let $ini(q)$ be the set of initial items for state q, derived from q as the smallest solution of

$$ini(q) = \{B \to .v \mid B \to v \wedge A \to \alpha.B\beta \in (q \cup ini(q))\}.$$

An alternative nonrecursive definition, similar to the definition of *predict*, is

$$ini(q) = \{B \to .v \mid B \to v \wedge A \to \alpha.\beta \in q \wedge \beta \Rightarrow^* B\gamma\}.$$

The transition function $goto: 2^{I_G} \times V \mapsto 2^{I_G}$ is defined by

$$goto(q, X) = \{A \to \alpha X.\beta \mid A \to \alpha.X\beta \in (q \cup ini(q))\}.$$

A recursive ascent recognizer is obtained by relating to each state q not only the above $[q]$, but also a function that we take to be the result of applying operator $\overline{[\cdot]}$ to the state:

$$\overline{[q]}: V \times N \mapsto 2^{I_G \times N}.$$

It has the specification

$$\overline{[q]}(X, i) = \{(A \to \alpha.\beta, j) \mid A \to \alpha.\beta \in q \wedge \beta \overset{*}{\Rightarrow} X\gamma \wedge \gamma \overset{*}{\to} x_{i+1} \ldots x_j\}.$$

Assuming, as before, that $x_{n+1} \notin V$, it is impossible that $A \to \alpha.\beta \in q \wedge \beta \overset{*}{\Rightarrow} x_{n+1}\gamma$; hence, $\overline{[q]}(x_{n+1}, n+1) = \emptyset$. For $i \leqslant n$ the above functions are recursively implemented by (Algorithm 2)

$$\begin{aligned}
[q](i) = {} & \overline{[q]}(x_{i+1}, i+1) \\
& \cup \{(A \to \alpha.\beta, j) \mid B \to .\varepsilon \in ini(q) \wedge (A \to \alpha.\beta, j) \in \overline{[q]}(B, i)\} \\
& \cup \{(A \to \alpha., i) \mid A \to \alpha. \in q\}, \qquad (2) \\
\overline{[q]}(X, i) = {} & \{(A \to \alpha.X\gamma, j) \mid A \to \alpha.X\gamma \in q \wedge (A \to \alpha X.\gamma, j) \in [goto(q, X)](i)\} \\
& \cup \{(A \to \alpha.\beta, j) \mid (A \to \alpha.\beta, j) \in \overline{[q]}(C, k) \wedge C \to .X\delta \in ini(q) \\
& \qquad \wedge (C \to X.\delta, k) \in [goto(q, X)](i)\}.
\end{aligned}$$

The correctness proof is similar to the one in Section 3, for the Earley parser. Moreover, it is a special case of the proof of the Marcus parsers that will be detailed in Section 5. A direct correctness proof can be found in [5, 7].

If the grammar is LR(0), one easily proves that each recognizer function for a canonical LR(0) state results in a set with at most one element. The functions for nonempty q may then be rephrased imperatively as

$[q](i)$: **if** $A\to\alpha.\in q$ **then** return $\{(A\to\alpha., i)\}$
else if $B\to.\varepsilon\in ini(q)$ **then** return $\overline{[q]}(B, i)$
else if $i<n$ **then** return $\overline{[q]}(x_{i+1}, i+1)$
else return $\emptyset$
fi

$\overline{[q]}(X, i)$: **if** $[goto(q, X)](i)=\emptyset$ **then** return $\emptyset$
else let $(A\to\alpha X.\beta, j)$ be the unique element of $[goto(q, X)](i)$.
if $A\to\alpha.X\beta\in q$ **then** return $\{(A\to\alpha.X\beta, j)\}$
else return $\overline{[q]}(A, j)$
fi
fi

Creating such functions for each LR(0) state, one obtains a deterministic LR(0) parser. In [7] it is explained how to improve its efficiency by replacing the functions by procedures that manipulate global variables.

Of course, the above nondeterministic LR(0) parser can also be used for arbitrary grammars. If the functions are memoized, the worst-case time complexity is $2^{|G|}O(n^3)$. In recent years it has become fashionable to consider LR parsers for parsing a natural language, and the above algorithm behaves better than the complicated Tomita [15] version of nondeterministic LR parsing [7]. Just like the Tomita parser and the above version of the Earley parser, the functional LR(0) parser does not terminate for cyclic grammars and for grammars for which there is a derivation $A\to^+\alpha A\beta$ such that $\alpha\to^+\varepsilon$. Just as for the Earley parser [6], however, lifting the ban on the use of ε-rules in leftmost symbol rewritings leads to an LR(0)-like parser that loops for cyclic grammars only.

5. Recursive ascent Marcus

Marcus [8] has suggested a type of look-ahead parsers that should mimic the processing of a natural language by humans. The characteristic assumption of a branch of linguistics is that Marcus' parser is the proper basis for processing a natural language in a deterministic way. Here we are not interested in such claims about natural language but focus on the main ideas of the parser itself. A problem is that Marcus parsers have not yet been formulated very accurately, although an

attempt has been made in [9]. The family of recognizers defined below constitute our formalization of Marcus' ideas about look-ahead. A formalization of idcas is rarely unique though, and we may have found one that deviates slightly from Marcus' own intentions. From the mathematical point of view, however, our family of parsers seems to be a natural one.

In the following we use a few new notations. By $k:\alpha$ we denote the k-prefix of α, with k a natural number. It is defined as follows: if $\alpha=\varepsilon$ or $k=0$ then $k:\alpha=\varepsilon$; if $\alpha\neq\varepsilon$ then $1:\alpha$ is the first symbol of α. More generally, if $\alpha\neq\varepsilon$ and $k>0$,

- if $1:\alpha\in V_T$ then $k:\alpha=1:\alpha$;
- if $1:\alpha\in V_N$ then
  if $k>|\alpha|$ then $k:\alpha=\alpha$
  otherwise $k:\alpha$ is the prefix of α with length k,

where $|\alpha|$ is the length of α. If the prefix $k:\alpha$ is removed from α, one is left with a postfix referred to as $\alpha:k$. We take the prefix and postfix operations to bind less tightly than concatenation. For instance, $k:\alpha\beta$ means the prefix of $\alpha\beta$.

The new family of parsers is based on a generalization of the notion of states. Whereas previously a state was a set of dotted grammar rules, it now becomes a set of objects $\gamma\rightarrow\alpha.\beta$, with $\gamma\in V_N V^*$ such that $1:\gamma$ rewrites in one step to a prefix of $\alpha\beta$. Correspondingly, we generalize some basic functions. Firstly, $ini(q)$ is (re)defined to be the smallest solution of

$$ini(q)=\{B\mu\rightarrow.\nu\mu \mid B\rightarrow\nu \wedge \gamma\rightarrow\alpha.B\beta\in(q\cup ini(q)) \wedge B\mu=k:B\beta\}$$

for some $k>0$. For example, if $A\rightarrow XBC$ and $B\rightarrow YZ$ are rules then

$$B\rightarrow.YZ\in ini(\{A\rightarrow X.BC\}) \text{ if } k=1,$$

$$BC\rightarrow.YZC\in ini(\{AD\rightarrow X.BCD\}) \text{ if } k=2,$$

$$BCD\rightarrow.YZCD\in ini(\{ADE\rightarrow X.BCDE\}) \text{ if } k=3, \text{ etc.}$$

Note how the right-hand sides of the items grow with k, and this, in fact, is the main idea: if a state with item $ADE\rightarrow XBCDE.$ is reached, a "reduce" action follows, implying that $A\rightarrow XBC$ applies. This decision is taken after having "looked ahead" the two symbols DE that follow the part of the input generated by A.

A nonrecursive definition of *ini* is possible with a new kind of rewriting: instead of elements of V^*, we rewrite elements of $V^*\times V^*$ with a family of rewriting relations denoted by $\Rightarrow_k$, with k a positive natural number. Their definition is

$$(A\beta,\gamma)\Rightarrow_k(k:\alpha\beta,\alpha\beta:k\gamma),$$

whenever $A\rightarrow\alpha$ is a grammar rule. For example, if $A\rightarrow BCD$ is a grammar rule, then

$$(A, YZ)\Rightarrow_1(B, CDYZ), \qquad (A, YZ)\Rightarrow_2(BC, DYZ),$$

$$(AX, YZ)\Rightarrow_2(BC, DXYZ), \qquad (AX, YZ)\Rightarrow_3(BCD, XYZ),$$

and if $A \to xCD$ is a rule, with x a terminal,

$$(A, YZ) \Rightarrow_k (x, CDYZ)$$

for $k = 1, 2, 3 \ldots$ The above function *ini* is definable in terms of $\Rightarrow_k$:

$$ini(q) = \{B\mu \to .\nu\mu \mid B \to \nu \wedge \gamma \to \alpha.\beta \in q \wedge (k:\beta, \beta:k) \overset{*}{\Rightarrow}_k (B\mu, \delta)\}.$$

Note that $A\alpha \Rightarrow X\beta \equiv (A, \alpha) \Rightarrow_1 (X, \beta)$ and both definitions of $ini(q)$ come down to the corresponding ones in Section 4, if one takes $k = 1$. Yet another way to define *ini* is

$$ini(q) = \{\mu \to .\delta\nu \mid \gamma \to \alpha.\beta \in q \wedge (k:\beta, \beta:k) \overset{*}{\Rightarrow}_k (\mu, \lambda) \wedge (\mu, \lambda) \Rightarrow_k (\delta, \nu\lambda)\}.$$

The function *goto* has to be generalized as well, turning its second argument into an element of V^+ instead of V:

$$goto(q, \delta) = \{\gamma \to \alpha\delta.\beta \mid \gamma \to \alpha.\delta\beta \in (q \cup ini(q)) \wedge \delta = k:\delta\beta\}.$$

Now consider recognition functions

$$[q](i) = \{(\gamma \to \alpha.\beta, j) \mid \gamma \to \alpha.\beta \in q \wedge \beta \overset{*}{\to} x_{i+1} \ldots x_j\},$$

$$\overline{[q]}(\delta, i) = \{(\gamma \to \alpha.\beta, j) \mid \gamma \to \alpha.\beta \in q \wedge (k:\beta, \beta:k) \overset{*}{\Rightarrow}_k (\delta, \lambda) \wedge \lambda \overset{*}{\to} x_{i+1} \ldots x_j\}.$$

For look-ahead parsers, it is customary to introduce a marker that signals the end of the input. We take $\perp$ for this marker, i.e. we require $x_{n+1} = \perp$. Then, if we define the initial state by $q_0 = \{S' \to .S \perp\}$, one has that $(S' \to .S \perp, n+1) \in [q_0](0)$ is equivalent to $S \overset{*}{\to} x_1 \ldots x_n$.

As the specifications of the recognition functions differ only slightly from the ones of Section 4, it will not come as a surprise that they can be implemented similarly (Algorithm 3):

$$\begin{aligned}
[q](i) = {} & \{(\gamma \to \alpha.\beta, j) \mid (\gamma \to \alpha.\beta, j) \in \overline{[q]}(x_{i+1}, i+1)\} \\
& \cup \{(\gamma \to \alpha.\beta, j) \mid B \to .\varepsilon \in ini(q) \wedge (\gamma \to \alpha.\beta, j) \in \overline{[q]}(B, i)\} \\
& \cup \{(\gamma \to \alpha., i) \mid \gamma \to \alpha. \in q\}, \qquad (3) \\
\overline{[q]}(\delta, i) = {} & \{(\gamma \to \alpha.\delta\lambda, j) \mid \gamma \to \alpha.\delta\lambda \in q \wedge (\gamma \to \alpha\delta.\lambda, j) \in [goto(q, \delta)](i)\} \\
& \cup \{(\gamma \to \alpha.\beta, j) \mid (\gamma \to \alpha.\beta, j) \in \overline{[q]}(\mu, l) \wedge \mu \to .\delta\nu \in ini(q) \\
& \qquad \wedge (\mu \to \delta.\nu, l) \in [goto(q, \delta)](i)\}.
\end{aligned}$$

Proof. The proof is isomorphic to the one in Section 3. First observe that

$$\beta \overset{*}{\to} x_{i+1} \ldots x_j \equiv \exists_\gamma((k:\beta, \beta:k) \overset{*}{\Rightarrow}_k (x_{i+1}, \gamma) \wedge \gamma \overset{*}{\to} x_{i+2} \ldots x_j)$$

$$\vee \exists_{B\gamma}((k:\beta, \beta:k) \overset{*}{\Rightarrow}_k (B, \gamma) \wedge B \to \varepsilon \wedge \gamma \overset{*}{\to} x_{i+1} \ldots x_j)$$

$$\vee (\beta = \varepsilon \wedge i = j),$$

which implies that

$$\begin{aligned}[q](i) = &\{(\gamma \to \alpha.\beta, j) \mid (\gamma \to \alpha.\beta, j) \in \overline{[q]}(x_{i+1}, i+1)\} \\ &\cup \{(\gamma \to \alpha.\beta, j) \mid B \to \varepsilon \wedge (\gamma \to \alpha.\beta, j) \in \overline{[q]}(B, i)\} \\ &\cup \{(\gamma \to \alpha., i) \mid \gamma \to \alpha. \in q\}.\end{aligned}$$

Because $\gamma \to \alpha.\beta \in q \wedge (k:\beta, \beta:k) \overset{*}{\Rightarrow}_k (B, \gamma)$ implies that all items $B \to .v$ are included in $ini(q)$, this is equivalent to the above version.

For establishing the correctness of $\overline{[q]}$ note that $(\beta_1, \beta_2) \overset{*}{\to}_k (\delta, \lambda)$ either consists of zero steps, in which case $\beta_1 = \delta$ and $\beta_2 = \lambda$, or contains at least one step:

$$\exists_\lambda((\beta_1, \beta_2) \overset{*}{\Rightarrow}_k (\delta, \lambda) \wedge \lambda \overset{*}{\to} x_{i+1} \ldots x_j) \equiv (\beta_1 = \delta \wedge \beta_2 \overset{*}{\to} x_{i+1} \ldots x_j)$$

$$\vee \exists_{\mu\nu\lambda l}((\beta_1, \beta_2) \overset{*}{\Rightarrow}_k (\mu, \lambda) \wedge (\mu, \lambda) \Rightarrow_k (\delta, \nu\lambda) \wedge \nu \overset{*}{\to} x_{i+1} \ldots x_l$$

$$\wedge \lambda \overset{*}{\to} x_{l+1} \ldots x_j).$$

We apply this equivalence for $\beta_1 = k:\beta$ and $\beta_2 = \beta:k$ to rewrite the specification of $\overline{[q]}$ as $\overline{[q]}(\delta, i) = S_0 \cup S_1$, with

$$S_0 = \{(\gamma \to \alpha.\delta\lambda, j) \mid \gamma \to \alpha.\delta\lambda \in q \wedge \delta = k:\delta\lambda \wedge \lambda \overset{*}{\to} x_{i+1} \ldots x_j\},$$

$$S_1 = \{(\gamma \to \alpha.\beta, j) \mid \gamma \to \alpha.\beta \in q \wedge (k:\beta, \beta:k) \overset{*}{\Rightarrow}_k (\mu, \lambda)$$

$$\wedge \mu \to .\delta\nu \in ini(q) \wedge \delta = k:\delta\nu \wedge \nu \overset{*}{\to} x_{i+1} \ldots x_l \wedge \lambda \overset{*}{\to} x_{l+1} \ldots x_j\}.$$

By the definition of *goto*, if $\gamma \to \alpha.\delta\lambda \in q$ and $\delta = k:\delta\lambda$ then $\gamma \to \alpha\delta.\lambda \in goto(q, \delta)$. Hence, with the specification of $[q]$, S_0 may be rewritten as

$$S_0 = \{(\gamma \to \alpha.\delta\lambda, j) \mid \gamma \to \alpha.\delta\lambda \in q \wedge (\gamma \to \alpha\delta.\lambda, j) \in [goto(q, \delta)](i)\}.$$

The set S_1 may be rewritten using the specification of $\overline{[q]}(\mu, l)$:

$$S_1 = \{(\gamma \to \alpha.\beta, j) \mid (\gamma \to \alpha.\beta, j) \in \overline{[q]}(\mu, l) \wedge \mu \to .\delta\nu \in ini(q)$$

$$\wedge \delta = k:\delta\nu \wedge \nu \overset{*}{\to} x_{i+1} \ldots x_l\}.$$

The existence of $\mu\rightarrow.\delta v$ in $ini(q)$ implies $\mu\rightarrow\delta.v\in goto(q,\delta)$ because $\delta=k:\delta v$. Hence,

$$S_1=\{(\gamma\rightarrow\alpha.\beta,j)\mid(\gamma\rightarrow\alpha.\beta,j)\in\overline{[q]}(\mu,l)\wedge\mu\rightarrow.\delta v\in ini(q)$$

$$\wedge(\mu\rightarrow\delta.v,l)\in[goto(q,\delta)](i)\}.\qquad\square$$

The fundamental reason for having the new states is that the items have longer right-hand sides, so that is will occur less often that the right-hand side of some item is a suffix of another one in the same state. As a consequence, the parser suffers from fewer reduce–reduce conflicts and is deterministic for more grammars. The look-ahead size k can be tuned to the grammar and may vary from state to state. Choosing $k=1$ for every state one recovers the parser of Section 4. Whereas in LR(k) parsers the look-ahead consists of k terminals, with k fixed, in the Marcus parser it consists of at most $k-1$ elements of V. This, in general, corresponds to an unbounded look-ahead in terms of terminals. For any value of k, however, the Marcus parser look-ahead may be 0 elements of V for some reductions. Also, when there are ε-rules, an element of V may derive 0 terminals. Hence, a finite look-ahead in terms of nonterminals may vanish in terms of terminals. It is, therefore, difficult to compare LR(k) parsers and Marcus parsers exactly. An interesting subject for future research would be to characterize the class of grammars that can be parsed deterministically with a Marcus parser.

6. Conclusions

The functional approach to LR parsing provides a high-level view on the subject compared to the standard theory. It might appear at first sight that this paper belongs to the theoretical realm only, and the formulation of the Algorithms (1–3) may seem esoteric, especially to people who are not used to functional formulations of algorithms. Nevertheless, we claim that the above is important in practice. In fact, the functional implementations need not be less efficient than conventional ones, especially if the functions are formulated in low-level imperative languages with efficient function calls, like C. This does not mean that efficiency considerations may not considerably alter the low-level realization of the functions. In [7] it is shown how to replace functions by procedures without arguments, if the parser is deterministic. If one wishes, one can go further and implement everything in assembly. Then the overhead of procedure calls may be eliminated altogether by replacing them by jump-to-subroutine instructions. Then one gets implementations like Pennello's [10], which (to our knowledge), in fact, is the first recursive ascent implementation of LR parsing (avant la lettre). Pennello invented his technique by looking at efficient implementations of recursive descent parsers, using his (recursive ascent) intuition about what happens in an LR parser. Had the above been the standard theory of LR parsing, his discovery would have been quite straightforward. The relation between

our theory and Pennello's implementation illustrates that some things that traditionally play an important role in the theory of LR parsing, such as stack manipulations, belong to the realm of implementation details. Another strategy that is possible to improve efficiency is the combining of recursive descent and ascent techniques [13].

Whereas the LR(0) parser of this paper is functionally equivalent to standard implementations, the recursive ascent Earley parsers of this paper and that of [6] are not. On the one hand, our functional implementations have problems with cyclic grammars. On the other, they allow the grammar to be compiled into a parser, and this definitely improves the parser's speed of execution. What we called an Earley parser here could also be called a non-deterministic PLR(0) parser [14] (PLR stands for *predictive* LR). Just like a PLR(0) parser, Algorithm 1 can be seen as a recursive descent parser on a transformed grammar. Moreover, the grammar transformations are essentially the same.

Marcus look-ahead parsers as formulated in this paper are so natural from the point of view of LR parsing that one would really hope that they will indeed prove to be natural in the linguistic sense as well. In any case, this paper's formalization of Marcus' ideas should be helpful to the linguistic application.

Acknowledgment

I am particularly indebted to two of my colleagues at Philips Research. Frans Kruseman Aretz invented recursive ascent parsing and took the time to comment on my ideas when they were still very immature. Lex Augusteijn pointed out to me the relevance of memo-functions and suggested the notion of pair rewriting used in Section 5. I also thank an anonymous referee for his useful suggestions.

References

[1] D.T. Barnard and J.R. Cordy, SL parses the LR languages, *Comput. Lang.* **13** (2) (1988) 65–74.
[2] J.C. Earley, An efficient context-free parsing algorithm, *Comm. ACM* **13** (2) (1970) 94–102.
[3] R.N. Horspool, Recursive ascent–descent parsers, in: D. Hammer, ed., *Compiler Compilers*, Lecture Notes in Computer Science Vol. 477 (Springer, Berlin, 1991) 1–10.
[4] F.E.J. Kruseman Aretz, On a recursive ascent parser, *Inform. Process. Lett.* **29** (1988) 201–206.
[5] R. Leermakers, Non-deterministic recursive ascent parsing, in: *Proc. 5th Conf. of the European Chapter of the Association for Computational Linguistics* (Berlin, 1991) 63–68.
[6] R. Leermakers, A recursive ascent Earley parser, *Inform. Process. Lett.* **41** (1992) 87–91.
[7] R. Leermakers, L. Augusteijn and F.E.J. Kruseman Aretz, A functional LR parser, *Theoret. Comput. Sci.* **104** (1992) 313–323.
[8] M.P. Marcus, *A Theory of Syntactic Recognition of Natural Language* (MIT Press, Cambridge, MA, 1980).
[9] R. Nozohoor-Farshi, On formalizations of Marcus' parser, in: *COLING '86* (Proceedings of the 11th International Conference on Computational Linguistics, University of Bonn, 1986) 533–535.
[10] T.J. Pennello, Very fast LR parsing, *SIGPLAN Notices* **21**(7) (1986) 145–151.

[11] G.H. Roberts, Recursive ascent: an LR analog to recursive descent, *SIGPLAN Notices* **23** (8) (1988) 23–29.
[12] G.H. Roberts, Another note on recursive ascent, *Inform. Process. Lett.* **32** (1989) 263–266.
[13] G.H. Roberts, From recursive ascent to recursive descent via compiler optimizations, *SIGPLAN Notices* **25** (4) (1990) 83–89.
[14] S. Sippu and E. Soisalon-Soininen, *Parsing Theory, Vol. II* (Springer, Berlin, 1990) 265–274.
[15] M. Tomita, *Efficient Parsing for Natural Language, A Fast Algorithm for Practical Systems* (Kluwer, Dordrecht, 1986).