# A recursive ascent Earley parser

René Leermakers

Instituut voor Perceptie Onderzoek,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
E-mail:leermake@prl.philips.nl

## 1 Introduction

Recently, the theory of LR-parsing gained new impetus by the discovery of the recursive ascent implementation technique for deterministic [2] and nondeterministic [3,4] 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.

One of the major application areas of the Earley algorithm [1] is the field of computational linguistics. A few years ago, Tomita [5] has proposed a nondeterministic LR-parser to parse natural languages. This parser has stirred some research to establish the differences between the Earley and Tomita parsers. One of the results of this research was the insight that the Earley algorithm can be seen as a shift-reduce parser and as such fits in a large family of parsers to which also nondeterministic LR-parsers belong. A consequence of this insight is that implementation techniques for LR-parsers can also be applied to the Earley algorithm. In particular, this can be done with the recursive ascent technique, and this is the subject of this note.

## 2 The algorithm

Consider CF grammar $G$, with terminals $V_T$ and nonterminals $V_N$. Let $V = V_N \cup V_T$. Recursive ascent parsers consist of a number of functions. Let us start with associating a function to each item. Items, grammar rules with a dot somewet referhere in the right hand side, are used ubiquitously in parsing theory to denote a partially recognized rule. A typical item is

written as $A \rightarrow \alpha.\beta$, with greek letters for arbitrary elements of $V^*$. We use the unorthodox embracing operator $[\cdot]$ to map each item to its function:

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

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

$$[A \rightarrow \alpha.\beta](i) = \{j | \beta \stackrel{*}{\rightarrow} x_{i+1}...x_j\},$$

with $x_1...x_n$ the string to be parsed. So, the function reports which parts of the string can be derived from $\beta$ starting from position $i$. Below we will find a constructive definition for this function that can be viewed as a functional implementation. If we add a grammar rule $S' \rightarrow S$ to $G$, with $S' \notin V$ then $S \stackrel{*}{\rightarrow} x_1...x_n$ is equivalent to $n \in [S' \rightarrow .S](0)$, so that the algorithm is to be invocated by calling $[S' \rightarrow .S](0)$.

To be able to construct an implementation of $[A \rightarrow \alpha.\beta]$ that has no problems with left-recursive grammars, we need so-called predict sets. Let $predict(A \rightarrow \alpha.\beta)$ be the set of initial items, that are derived from $A \rightarrow \alpha.\beta$ by the closure operation:

$$predict(A \rightarrow \alpha.\beta) = \{B \rightarrow .\mu | B \rightarrow \mu \wedge \beta \stackrel{*}{\Rightarrow} B\gamma\}.$$

The double arrow $\Rightarrow$ denotes a left-most-symbol rewriting with a non-$\epsilon$ grammar rule, i.e.

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

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

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

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

$$\overline{[A \rightarrow \alpha.\beta]}(X, i) = \{j | \exists_\gamma : \beta \stackrel{*}{\Rightarrow} X\gamma \wedge \gamma \stackrel{*}{\rightarrow} x_{i+1}...x_j\}.$$

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

$$[A \to \alpha.\beta](i) = \overline{[A \to \alpha.\beta]}(x_{i+1}, i+1) \cup$$
$$\{j | \exists_B : B \to .\epsilon \in predict(A \to \alpha.\beta) \wedge j \in \overline{[A \to \alpha.\beta]}(B, i)\} \cup$$
$$\{i | \beta = \epsilon\}$$

$$\overline{[A \to \alpha.\beta]}(X, i) = \{j | \exists_\gamma : \beta = X\gamma \wedge j \in [A \to \alpha X.\gamma](i)\} \cup$$
$$\{j | \exists_{C\delta k} : j \in \overline{[A \to \alpha.\beta]}(C, k) \wedge C \to .X\delta \in predict(A \to \alpha.\beta) \wedge$$
$$k \in [C \to X.\delta](i)\}$$

The correctness of this implementation is shown by the following proof: First we notice that

$$\beta \xrightarrow{*} x_{i+1}...x_j \equiv \exists_\gamma(\beta \overset{*}{\Rightarrow} x_{i+1}\gamma \wedge \gamma \xrightarrow{*} x_{i+2}...x_j) \vee$$
$$\exists_{B\gamma}(\beta \overset{*}{\Rightarrow} B\gamma \wedge B \to \epsilon \wedge \gamma \xrightarrow{*} x_{i+1}...x_j) \vee$$
$$(\beta = \epsilon \wedge i = j).$$

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

$$[A \to \alpha.\beta](i) = \{j | \exists_\gamma : \beta \overset{*}{\Rightarrow} x_{i+1}\gamma \wedge \gamma \xrightarrow{*} x_{i+2}...x_j\} \cup$$
$$\{j | \exists_{B\gamma} : B \to \epsilon \wedge \beta \overset{*}{\Rightarrow} B\gamma \wedge \gamma \xrightarrow{*} x_{i+1}...x_j\} \cup$$
$$\{j | \beta = \epsilon \wedge i = j\}.$$

This directly leads to the implementation given above, using the specifications of $\overline{[A \to \alpha.\beta]}(x_{i+1}, i+1)$, $\overline{[A \to \alpha.\beta]}(B, i)$ and $predict(A \to \alpha.\beta)$.

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

$$\exists_\gamma(\beta \overset{*}{\Rightarrow} X\gamma \wedge \gamma \xrightarrow{*} x_{i+1}...x_j) \equiv \exists_\gamma(\beta = X\gamma \wedge \gamma \xrightarrow{*} x_{i+1}...x_j) \vee$$
$$\exists_{C\delta\gamma k}(\beta \overset{*}{\Rightarrow} C\gamma \wedge C \to X\delta \wedge \delta \xrightarrow{*} x_{i+1}...x_k \wedge \gamma \xrightarrow{*} x_{k+1}...x_j)$$

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

$$S_0 = \{j | \exists_\gamma : \beta = X\gamma \wedge \gamma \xrightarrow{*} x_{i+1}...x_j\}$$

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

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

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

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

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

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

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

□

# 3 Complexity and Variants

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 [1]. The worst-case complexity analysis of the memo-ized 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 (bounded by the number of nonterminals), 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 $\overline{[I]}$ invocates $O(qn)$ 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 $q$ slightly differs. Worst-case complexities need not be relevant in practice. We claim that for many practical grammars the present algorithm is more efficient than existing implementations, for the following reason. The above formulae can be interpreted as to define 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 $A \xrightarrow{+} A$, 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 $\epsilon$. This occurs if for some $k$ one has that for $i = 1...k$

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

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 $A \xrightarrow{+} \alpha A \beta$ such that $\alpha \xrightarrow{+} \epsilon$. 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 left most symbol derivation $\Rightarrow$ may not use $\epsilon$-rules. Then this paper's analysis can be repeated. The redefinition of $\Rightarrow$ affects the function *predict* and the way $\beta \xrightarrow{*} x_{i+1}...x_j$ is to be expressed in terms of $\Rightarrow$:

$$\beta \xrightarrow{*} x_{i+1}...x_j \equiv \exists_\gamma(\beta \stackrel{*}{\Rightarrow} x_{i+1}\gamma \wedge \gamma \xrightarrow{*} x_{i+2}...x_j) \vee (\beta \xrightarrow{*} \epsilon \wedge i = j).$$

Also the decomposition of $\beta \stackrel{*}{\Rightarrow} X\gamma$ must be rephrased: $X$ is introduced by a grammar rule $C \rightarrow \mu X \delta$ or it is not. The result is the following recognizer:

$$[A \rightarrow \alpha.\beta](i) = \overline{[A \rightarrow \alpha.\beta]}(x_{i+1}, i+1) \cup \{i | \beta \xrightarrow{*} \epsilon\}$$

$$\begin{aligned}\overline{[A \rightarrow \alpha.\beta]}(X,i) = {} & \{j | \exists_{\gamma\mu} : \beta = \mu X\gamma \wedge \mu \xrightarrow{*} \epsilon \wedge j \in [A \rightarrow \alpha\mu X.\gamma](i)\} \cup \\ & \{j | \exists_{C\delta\mu k} : j \in \overline{[A \rightarrow \alpha.\beta]}(C,k) \wedge C \rightarrow .\mu X\delta \in predict(A \rightarrow \alpha.\beta) \wedge \\ & \mu \xrightarrow{*} \epsilon \wedge k \in [C \rightarrow \mu X.\delta](i)\}\end{aligned}$$

To conclude, the above offers attractive alternatives to the standard Earley parser. Only for cyclic grammars there is a problem with termination. For other grammars one gains efficiency. The above recognition algorithms are not the most efficient ones, however. For instance, there is an additional improvement following from the observation that $[A \rightarrow \alpha.\beta]$ and $\overline{[A \rightarrow \alpha.\beta]}$ only depend on $\beta$. Therefore, functions for different items may be identical

and can be identified. A similar improvement is applicable to many implementations of the Earley algorithm. A recursive ascent recognizer with this improvement is

$$[\beta](i) = \overline{[\beta]}(x_{i+1}, i+1) \cup \{i | \beta \xrightarrow{*} \epsilon\}$$

$$\overline{[\beta]}(X, i) = \{j | \exists_{\gamma\mu} : \beta = \mu X \gamma \wedge \mu \xrightarrow{*} \epsilon \wedge j \in [\gamma](i)\} \cup$$
$$\{j | \exists_{C\delta\mu k} : j \in \overline{[\beta]}(C, k) \wedge C \rightarrow .\mu X \delta \in predict(\beta) \wedge \mu \xrightarrow{*} \epsilon \wedge k \in [\delta](i)\}$$

with $predict(\beta) = \{B \rightarrow .\mu | B \rightarrow \mu \wedge \beta \xRightarrow{*} B\gamma\}$. The algorithm is to be invocated by calling $[S](0)$.

## References


1 J.C. Earley, An efficient context-free parsing algorithm, *Commun. ACM* (1970) 13(2):94-102.

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

3 R. Leermakers, Non-deterministic Recursive Ascent Parsing, Proceedings of the 5th Conference of the European Chapter of the Association of Computational Linguistics, 1991.

4 R. Leermakers, L. Augusteijn and F.E.J. Kruseman Aretz, A functional LR-parser, *Theoretical Computer Science* (accepted).

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