Non-deterministic Recursive Ascent Parsing

René Leermakers

Philips Research Laboratories,
P.O. Box 80.000, 5600 JA Eindhoven, The Netherlands

E-mail:leermake@rosetta.prl.philips.nl

ABSTRACT

A purely functional implementation of LR-parsers is given, together with a simple correctness proof. It is presented as a generalization of the recursive descent parser. For non-LR grammars the time-complexity of our parser is cubic if the functions that constitute the parser are implemented as memo-functions, i.e. functions that memorize the results of previous invocations. Memo-functions also facilitate a simple way to construct a very compact representation of the parse forest. For LR(0) grammars, our algorithm is closely related to the recursive ascent parsers recently discovered by Kruseman Aretz [1] and Roberts [2]. Extended CF grammars (grammars with regular expressions at the right hand side) can be parsed with a simple modification of the LR-parser for normal CF grammars.

1 Introduction

In this paper we give a purely functional implementation of LR-parsers, applicable to general CF grammars. It will be obtained as a generalization of the well-known recursive descent parsing technique. For LR(0) grammars, our result implies a deterministic parser that is closely related to the recursive ascent parsers discovered by Kruseman Aretz [1] and Roberts [2]. In the general non-deterministic case, the parser has cubic time complexity if the parse functions are implemented as memo-functions [3], which are functions that memorize and re-use the results of previous invocations. Memo-functions are easily implemented in most programming languages. The notion of memo-functions is also used to define an algorithm that constructs a cubic representation for the parse forest, i.e. the collection of parse trees.

It has been claimed by Tomita that non-deterministic LR-parsers are useful for natural language processing. In [4] he presented a discussion about how to do non-deterministic LR-parsing, with a device called a graph-structured stack. With our parser we show that no explicit stack manipulations are needed; they can be expressed implicitly with the use of appropriate programming language concepts.

Most textbooks on parsing do not include proper correctness proofs for LR-parsers, mainly because such proofs tend to be rather involved. The theory of LR-parsing should still be considered underdeveloped, for this reason. Our presentation, however, contains a surprisingly simple correctness proof. In fact, this proof is this paper's major contribution to parsing theory. One of its lessons is that the CF grammar class is often the natural one to proof parsers for, even if these parsers are devoted to some special class of grammars. If the grammar is restricted in some way, a parser for general CF grammars may have properties that enable smart implementation tricks to enhance efficiency. As we show below, the relation between LR-parsers and LR-grammars is of this kind.

Especially in natural language processing, standard CF grammars are often too limited in their strong generative power. The extended CF grammar formalism, allowing rules to have regular expressions at the right hand side, is a useful extension, for that reason. It is not difficult to generalize our parser to cope with extended grammars, although the application of LR-parsing to extended CF grammars is well-known to be problematic [5].

We first present the recursive descent recognizer in a way that allows the desired generalization. Then we obtain the recursive ascent recognizer and its proof. If the grammar is LR(0) a few implementation tricks lead to the recursive ascent recognizer of ref. [1]. Subsequently, the time and space complexities of the recognizer are analysed, and the algorithm for constructing a cubic representation for parse forests is given. The paper ends with a discussion of extended CF grammars.

2 Recursive descent

Consider CF grammar G, with terminals V_T and non-terminals 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 non-terminal. Here we present a variant that consists of functions, one for each item (dotted rule). We use the unorthodox embracing operator $[\cdot]$ to map each item to its function: (we use greek letters for arbitrary elements of V^*)

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

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

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

with $x_1 ... x_n$ the sentence to be parsed. A recursive implementation for these functions is given by ($b \in V_T, B \in V_N$)

$$[A \rightarrow \alpha.](i) = \{i\}$$
$$[A \rightarrow \alpha.b\gamma](i) = \{j | b = x_{i+1} \wedge j \in [A \rightarrow \alpha b.\gamma](i+1)\}$$
$$[A \rightarrow \alpha.B\gamma](i) = \\ \{j | k \in [B \rightarrow .\delta](i) \wedge j \in [A \rightarrow \alpha B.\gamma](k)\}$$

We keep to the custom of omitting existential quantification (here for k, δ) in definitions of this kind.

The proof is elementary and based on

$$\beta \rightarrow^* x_{i+1} ... x_j \equiv (\beta = \epsilon \wedge i = j) \vee \\ \exists_\gamma(\beta = x_{i+1}\gamma \wedge \gamma \rightarrow^* x_{i+2} ... x_j) \vee \\ \exists_{B\gamma\delta k}(\beta = B\gamma \wedge B \rightarrow \delta \wedge \delta \rightarrow^* x_{i+1} ... x_k \wedge \\ \gamma \rightarrow^* x_{k+1} ... x_j)$$

If we add a grammar rule $S' \rightarrow S$ to G, with $S' \notin V$ then $S \rightarrow^* x_1 ... x_n$ is equivalent to $n \in [S' \rightarrow .S](0)$.

The recursive descent recognizer works for any CF grammar except for grammars for which $\exists_{A\alpha\beta}(A \rightarrow \alpha \wedge \alpha \rightarrow^* A\beta)$. For such left-recursive grammars the recognizer does not terminate, as execution of $[A \rightarrow .\alpha](i)$ will lead to a call of itself. The recognition is not a linear process in general: the function calls $[A \rightarrow \alpha.B\gamma](i)$ lead to calls $[B \rightarrow .\delta](i)$ for all values of δ such that $B \rightarrow \delta$ is a grammar rule.

3 The ascent recognizer

One way to make the recognizer more deterministic is by combining 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 be an arbitrary state. We associate to each state q a function, re-using the above operator $[\cdot]$,

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

that meets the specification

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

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

Let $ini(q)$ be the set of initial items for state q, that are derived from q by the closure operation:

$$ini(q) = \{B \rightarrow .\lambda | B \rightarrow \lambda \wedge A \rightarrow \alpha.\beta \in q \wedge \beta \Rightarrow^* B\gamma\}.$$

The double arrow $\Rightarrow$ denotes a left-most-symbol rewriting $B\alpha \Rightarrow C\beta\alpha$, using a non-$\epsilon$ rule $B \rightarrow C\beta$. The transition function *goto* is defined by ($B \in V$)

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

Also define

$$pop(A \rightarrow \alpha B.\beta) = A \rightarrow \alpha.B\beta$$
$$lhs(A \rightarrow \alpha.\beta) = A$$
$$final(A \rightarrow \alpha.\beta) = (|\beta| = 0)$$

with $B \in V$, and $|\beta|$ the number of symbols in β (with $|\epsilon| = 0$). A recursive ascent recognizer may be 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:

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

It has the specification

$$\overline{[q]}(B, i) = \{(A \rightarrow \alpha.\beta, j) | A \rightarrow \alpha.\beta \in q \wedge \\ \beta \Rightarrow^* B\gamma \wedge \gamma \rightarrow^* x_{i+1} ... x_j\}$$

For $i \underline{>} n$ (n is the sentence length) it follows that $[q](i) = \overline{[q]}(B, i) = \emptyset$, whereas for $i \leq n$ the functions are recursively implemented by

$$[q](i) = \overline{[q]}(x_{i+1}, i+1) \cup \\ \{(I, j) | B \rightarrow .\epsilon \in ini(q) \wedge (I, j) \in \overline{[q]}(B, i)\} \cup \\ \{(I, i) | I \in q \wedge final(I)\}$$

$$\overline{[q]}(B, i) = \{(pop(I), j) | \\ (I, j) \in [goto(q, B)](i) \wedge pop(I) \in q\} \cup \\ \{(I, j) | (J, k) \in [goto(q, B)](i) \wedge \\ pop(J) \in ini(q) \wedge (I, j) \in \overline{[q]}(lhs(J), k)\}$$

Proof:
First we notice that

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

Hence

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

This is equivalent to the earlier version because we may replace the clause $B \rightarrow \epsilon$ by $B \rightarrow .\epsilon \in ini(q)$. Indeed, if state q has item $A \rightarrow \alpha.\beta$ and if there is a left-most-symbol derivation $\beta \Rightarrow^* B\gamma$ then all items $B \rightarrow .\lambda$ are included in $ini(q)$.

For establishing the correctness of $\overline{[q]}$ notice that $\beta \Rightarrow^* B\gamma$ either contains zero steps, in which case $\beta = B\gamma$, or it contains at least one step:

$$\exists_\gamma(\beta \Rightarrow^* B\gamma \wedge \gamma \rightarrow^* x_{i+1} ... x_j) \equiv \\ \exists_\gamma(\beta = B\gamma \wedge \gamma \rightarrow^* x_{i+1} ... x_j) \vee \\ \exists_{C\delta\gamma k}(\beta \Rightarrow^* C\gamma \wedge C \rightarrow B\delta \wedge \delta \rightarrow^* x_{i+1} ... x_k \wedge \gamma \rightarrow^* \\ x_{k+1} ... x_j)$$

Hence $\overline{[q]}(B, i)$ may be written as the union of two sets, $\overline{[q]}(B, i) = S_0 \cup S_1$:

$$S_0 = \{(A \rightarrow \alpha.B\gamma, j) | \\ A \rightarrow \alpha.B\gamma \in q \wedge \gamma \rightarrow^* x_{i+1} ... x_j\}$$

$$S_1 = \{(A \rightarrow \alpha.\beta, j) | A \rightarrow \alpha.\beta \in q \wedge \beta \Rightarrow^* C\gamma \wedge \\ C \rightarrow B\delta \wedge \delta \rightarrow^* x_{i+1} ... x_k \wedge \gamma \rightarrow^* x_{k+1} ... x_j\}.$$

By the definition of *goto*, if $A \rightarrow \alpha.B\gamma \in q$ then $A \rightarrow \alpha B.\gamma \in goto(q, B)$. Hence, with the specification of $[q]$, S_0 may be rewritten as

$$S_0 = \{(A \rightarrow \alpha.B\gamma, j) | A \rightarrow \alpha.B\gamma \in q \wedge \\ (A \rightarrow \alpha B.\gamma, j) \in [goto(q, B)](i)\}$$

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

$$S_1 = \{(A \rightarrow \alpha.\beta, j) | (A \rightarrow \alpha.\beta, j) \in \overline{[q]}(C,k) \wedge$$
$$C \rightarrow B\delta \wedge \delta \rightarrow^* x_{i+1}...x_k\}.$$

Also, as before, $\beta \Rightarrow^* C\gamma$ implies that all items $C \rightarrow .\lambda$ are in $ini(q)$, and the existence of $C \rightarrow .B\delta$ in $ini(q)$ implies $C \rightarrow B.\delta \in goto(q, B)$:

$$S_1 = \{(A \rightarrow \alpha.\beta, j) | (A \rightarrow \alpha.\beta, j) \in \overline{[q]}(C,k) \wedge$$
$$C \rightarrow .B\delta \in ini(q) \wedge$$
$$(C \rightarrow B.\delta, k) \in [goto(q,B)](i)\}.$$

□

In the computation of $[q_0](0)$, functions are needed only for states in the canonical collection of LR(0) states [6] for G, i.e. for every state that can be reached from the initial state by repeated application of the *goto* function. Note that in general the state $\emptyset$ will be among these, and that both $[\emptyset](i)$ and $\overline{[\emptyset]}(B,i)$ are empty sets for all $i \geq 0$ and $B \in V$.

4 Deterministic variants

One can prove that, if the grammar is LR(0), each recognizer function for a canonical LR(0) state results in a set with at most one element. The functions for non-empty q may in this case be rephrased as

$[q](i)$:

```
if, for some $I$, $I \in q \wedge final(I)$ then return $\{(I,i)\}$ else
if $B \rightarrow .\epsilon \in ini(q)$ then return $\overline{[q]}(B,i)$
else if $i < n$ then return $\overline{[q]}(x_{i+1}, i+1)$
else return $\emptyset$
fi
```

$\overline{[q]}(B,i)$:

```
if $[goto(q,B)](i) = \emptyset$ then return $\emptyset$ else
let $(I,j)$ be the unique element of $[goto(q,B)](i)$. Then:
    if $pop(I) \in q$ then return $\{(pop(I),j)\}$
    else return $\overline{[q]}(lhs(I),j)$
    fi
fi
```

Reversely, the implementations of $[q](i)$ and $\overline{[q]}(B,i)$ of the previous section can be seen as non-deterministic versions of the present formulation, which therefore provides an intuitive picture that may be helpful to understand the non-deterministic parsing process in an operational way.

Each function can be replaced by a procedure that, instead of returning a function result, assigns the result to a global (set) variable. As this set variable may contain at most one element, it can be represented by three variables, a boolean b, an item R and an integer i. If a function would have resulted in the set $\{(I,j)\}$, the global variables are set to $b = TRUE$, $R = I$ and $i = j$. A function value $\emptyset$ is represented by $b = FALSE$. Also the arguments of the functions are superfluous now. The rôle of argument i can be played by the global variable with the same name, and $lhs(R)$ can be used instead of argument B of $\overline{[q]}$. Consequently, procedure $[\emptyset]$ becomes a statement $b := FALSE$, whereas for non-empty q one gets the procedures (keeping the names $[q]$ and $\overline{[q]}$, trusting no confusion will arise):

$[q]$:

```
if, for some $I$, $I \in q \wedge final(I)$ then $R := I$
else if $B \rightarrow .\epsilon \in ini(q)$ then $R := B \rightarrow \epsilon.$; $\overline{[q]}$
else if $i < n$ then $R := x_{i+1} \rightarrow x_{i+1}.$; $i := i+1$; $\overline{[q]}$
else $b := FALSE$
fi
```

$\overline{[q]}$:

```
$[goto(q, lhs(R))]$;
if $b$ then
    if $pop(R) \in q$ then $R := pop(R)$
    else $\overline{[q]}$
    fi
fi
```

Note that these procedures do not depend on the details of the right hand side of R. Only the number of symbols before the dot is relevant for the test "$pop(R) \in q$". Therefore, R can be replaced by two variables $X \in V$ and an integer l, making the following substitutions in the previous procedures:

$R := A \rightarrow \alpha.$	$\Rightarrow$	$X := A; l := \|\alpha\|$
$R := pop(R)$	$\Rightarrow$	$l := l - 1$
$pop(R) \in q$	$\Rightarrow$	$l \neq 1 \vee X = S'$
$lhs(R)$	$\Rightarrow$	X

After these substitutions, one gets close to the recursive ascent recognizer as it was presented in [1]. A recognizer that is virtually the same as in [1] is obtained by replacing the tail-recursive procedure $\overline{[q]}$ by an iterative loop. Then one is left with one procedure for each state. While parsing there is, at each instance, a stack of activated procedures that corresponds to the stacks that are explicitly maintained in conventional implementations of deterministic LR-parsers.

5 Complexity

For LL(0) grammars the recursive descent recognizer is deterministic and works in linear time. The same is true of the ascent recognizer for LR(0) grammars. In the general, non-deterministic, case the recursive descent and ascent recognizers need exponential time unless the functions are implemented as memo-functions [3]. 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. Devices like graph-structured stacks [4], parse matrices [7], or well-formed substring tables [8], are in fact low-level realizations of the abstract notion of memo-functions. The complexity analysis of the recognizers is quite simple. There are $O(n)$ different invocations of parser functions. The functions call at most $O(n)$ other functions, that all result in a set with $O(n)$ elements (note that there exist only $O(n)$ pairs (I, j) with $I \in I_G, i \leq j \leq n$). Merging these sets to one set with no duplicates can be accomplished in $O(n^2)$ time on a random access machine. Hence, the total time-complexity is $O(n^3)$. The space needed for storing function results is $O(n)$ per invocation, i.e. $O(n^2)$ for the whole recognizer.

The above considerations only hold if the parser terminates. The recursive descent parser terminates for all grammars that are not left-recursive. For the recursive ascent parser, the situation is more complicated. If the grammar has a cyclic derivation $B \rightarrow^* B$, the execution of $\overline{[q]}(B, i)$ leads to a call of itself. Also, there may be a cycle of transitions labeled by non-terminals that derive ϵ, e.g. if $goto(q, B) = q \wedge B \rightarrow \epsilon$, so that the execution of $[q](i)$ leads to a call of itself. There are non-cyclic grammars that suffer from such a cycle (e.g. $S \rightarrow SSb$, $S \rightarrow \epsilon$). Hence, the ascent parser does not terminate if the grammar is cyclic or if it leads to a cycle of transitions labeled by non-terminals that derive ϵ. Otherwise, execution of $\overline{[q]}(B, i)$ can only lead to calls of $[p](i)$ with $p \neq q$ and to calls of $\overline{[q]}(C, k)$, such that either $k > i$ or $C \rightarrow^* B \wedge C \neq B$. As there are only finitely many such p, C, the parser terminates. Note that both the recursive descent and ascent recognizer terminate for any grammar, if the recognizer functions are implemented as memo-functions with the property that a call of a function with some arguments yields $\emptyset$ while it is under execution. For instance, if execution of $[q](i)$ leads to a call of itself, the second call is to yield $\emptyset$. A remark of this kind, for the recursive descent parser, was first made in ref. [8]. The recursive descent parser then becomes virtually equivalent to a version of the standard Earley algorithm [9] that stores items $A \rightarrow \alpha.\beta$ in parse matrix entry T_{ij} if $\beta \rightarrow^* x_{i+1}...x_j$, instead of storing it if $\alpha \rightarrow^* x_{i+1}...x_j$.

The space required for a parser that also calculates a parse forest, is dominated by this forest. We show in the next section that it may be compressed into a cubic amount of space. In the complexity domain our ascent parser beats its rival, Tomita's parsing method [4], which is non-polynomial: for each integer k there exists a grammar such that the complexity of the Tomita parser is worse than n^k.

In addition to the complexity as a function of sentence length, one may also consider the complexity as a function of grammar size. It is clear that both time and space complexity are proportional to the number of parsing procedures. The number of procedures of the recursive descent parser is proportional to the number of items, and hence a linear function of the grammar size. The recursive ascent parser, however, contains two functions for each LR-state and is hence proportional to the size of the canonical collection of LR(0) states. In the worst case, this size is an exponential function of grammar size, but in the average natural language case there seems to be a linear, or even sublinear, dependence [4].

6 Parse forest

Usually, the recognition process is followed by the construction of parse trees. For ambiguous grammars, it becomes an issue how to represent the set of parse trees as compactly as possible. Below, we describe how to obtain a cubic representation in cubic time. We do so in three steps.

In the first step, we observe that ambiguity often arises locally: given a certain context $C[\cdot]$, there might be several parse subtrees $t_1...t_k$ (all deriving the same substring $x_{i+1}...x_j$ from the same symbol A) that fit in that same context, leading to the parse trees $C[t_1]$, $C[t_2]$, ... ,$C[t_k]$ for the given string $x_1...x_n$. Instead of representing these parse trees separately, repeating each time the context C, we can represent them collectively as $C[\{t_1, ..., t_k\}]$. Of course, this idea should be applied recursively. Technically, this leads to a kind of tree-like structure in which each child is a set of substructures rather than a single one.

The sharing of context can be carried one step further. If we have, in one and the same context, a number of applied occurrences of a production rule $A \rightarrow \alpha\beta$ which share also the same parse forest for α, we can represent the context of $A \rightarrow \alpha\beta$ itself and the common parse forest for α only once and fit the set of parse forests for β into that. Again this idea has to be applied recursively. Technically, this leads to a binary representation of parse trees, with each node having at most two sons, and to the application of the context sharing technique to this binary representation.

These two ideas are captured by introducing a function f with the interpretation that $f(\beta, i, j)$ represents the parse forest of all derivations from $\beta \in V^*$ to $x_{i+1}...x_j$, for all i, j such that $0 \leq i \leq j \leq n$. The following recursive definitions fix the parse forest representation formally:

$f(\epsilon, i, j) = \{[]|i = j\}$,

$f(a, i, j) = \{a|j = i + 1 \wedge x_{i+1} = a\}$, for all $a \in V_T$,

$f(A, i, j) = \{(A, f(\alpha, i, j))|A \rightarrow \alpha \wedge$
$\alpha \rightarrow^* x_{i+1}...x_j\}$, for all $A \in V_N$,

$f(AB\beta, i, j) = \{(f(A, i, k), f(B\beta, k, j))|$
$i \leq k \leq j \wedge A \rightarrow^* x_{i+1}...x_k \wedge B\beta \rightarrow^* x_{k+1}...x_j\}$, for all $A, B \in V$.

The representation for the set of parse trees is then just $f(S, 0, n)$.

We now come to our third step. Suppose, for the moment, that the guards $\alpha \rightarrow^* x_{i+1}...x_j$ and the like, occurring above, can be evaluated in some way or another. Then we can use function f to *compute* the representation of the set of parse trees for sentence $x_1...x_n$. If we make use of memo-functions to avoid repeated computation of a function applied to the same arguments, we see that there are at most $O(n^2)$ function evaluations.

If we represent function values by *references* to the set representations rather than by the sets themselves, the most complicated function evaluation consumes an additional amount of storage that is $O(n)$: for $j-i+1$ values of k we have to perform the construction of a pair of (copies of) two references, costing a unit amount of storage each. Therefore, the total amount of space needed for the representation of all parse trees is $O(n^3)$.

The evaluation of the guards $\alpha \rightarrow^* x_{i+1}...x_j$ etc. amounts exactly to solving a collection of recognition problems. Note that a top-down parser is possible that merges the recognition and tree-building phases, by writing

$f(A,i,j) = \{(A, f(\alpha,i,j)) | A \rightarrow \alpha \wedge f(\alpha,i,j) \neq \emptyset\}$, for all $A \in V_N$,

$f(AB\beta,i,j) = \{(f(A,i,k), f(B\beta,k,j)) |$
$i \leq k \leq j \wedge f(A,i,k) \neq \emptyset \wedge f(B\beta,k,j) \neq \emptyset\}$,
for all $A, B \in V$,

the other cases for f being left unchanged. Note the similarity between the recognizing part of this algorithm and the descent recognizer of section 2. Again, this parser is a cubic algorithm if we use memo-functions.

Another approach is to apply a bottom-up recognizer first and derive from it a set P containing triples (β,i,j) only if $\beta \rightarrow^* x_{i+1}...x_j$, and at least those triples (β,i,j) for which the guards $\beta \rightarrow^* x_{i+1}...x_j$ are evaluated during the computation of $f(S,0,n)$ (i.e., for each derivation $S \rightarrow^* x_1...x_kAx_{j+1}...x_n \rightarrow x_1...x_k\alpha\beta x_{j+1}...x_n \rightarrow^* x_1...x_i\beta x_{j+1}...x_n \rightarrow^* x_1...x_n$, the triples (β,i,j) and (A,k,j) should be in P). The simplest way to obtain such P from our recognizer is to assume an implementation of memo-functions that enables access to the memoized function results, after executing $[q_0](0)$. Then one has the disposal of the set

$\{(\beta,i,j) | [q](i)$ was invocated and
$(A \rightarrow \alpha.\beta, j) \in [q](i)\}$

Clearly, (β,i,j) is only in this set if $\beta \rightarrow^* x_{i+1}...x_j$. Note, however, that no pairs $(A \rightarrow .\beta, j)$ are included in $[q](i)$ (except if $A = S'$). We remedy this with a slight change of the specifications of $[q]$ and $[\overline{q}]$, defining $\overline{q} \equiv q \cup ini(q)$:

$[q](i) =$
$\{(A \rightarrow \alpha.\beta, j) | A \rightarrow \alpha.\beta \in \overline{q} \wedge \beta \rightarrow^* x_{i+1}...x_j\}$

$[\overline{q}](B,i) = \{(A \rightarrow \alpha.\beta, j) | A \rightarrow \alpha.\beta \in \overline{q} \wedge$
$\beta \Rightarrow^* B\gamma \wedge \gamma \rightarrow^* x_{i+1}...x_j\}$

A recursive implementation of the recognition functions now is

$[q](i) = \{(I,j) | (I,j) \in [\overline{q}](x_{i+1}, i+1)\} \cup$
$\{(I,j) | B \rightarrow .\epsilon \in ini(q) \wedge (I,j) \in [\overline{q}](B,i)\} \cup$
$\{(I,i) | I \in \overline{q} \wedge final(I)\}$

$[\overline{q}](B,i) = \{(pop(I),j) | (I,j) \in [goto(q,B)](i)\} \cup$
$\{(I,j) | (J,k) \in [goto(q,B)](i) \wedge$
$pop(J) \in ini(q) \wedge (I,j) \in [\overline{q}](lhs(J),k)\}$

If we define, for this revised recognizer,

$P = \{(\beta,i,j) | [q](i)$ was invocated and
$(A \rightarrow \alpha.\beta, j) \in [q](i)\} \cup$
$\{(A,i,j) | [q](i)$ was invocated and
$(A \rightarrow .\beta, j) \in [q](i)\} \cup$
$\{(x_{i+1}, i, i+1) | 0 \leq i < n\}$,

it contains all triples that are needed in $f(S,0,n)$, and we may write the forest constructing function as

$f(A,i,j) = \{(A, f(\alpha,i,j)) | A \rightarrow \alpha \wedge (\alpha,i,j) \in P\}$, for all $A \in V_N$,

$f(AB\beta,i,j) = \{(f(A,i,k), f(B\beta,k,j)) |$
$(A,i,k) \in P \wedge (B\beta,k,j) \in P\}$, for all $A, B \in V$,

the other cases for f being left unchanged again. There exists a representation of P in quadratic space such that the presence or absence of an arbitrary triple can be decided upon in unit time. As a result, the time complexity of $f(S,0,n)$ is cubic.

7 Extended CF grammars

An extended CF grammar consists of grammar rules with regular expressions at the right hand side. Every extended CF grammar can be translated into a normal CF grammar by replacing each right hand side by a regular (sub)grammar. The strong generative power is different from CF grammars, however, as the degree of the nodes in a derivation tree is unbounded. To apply our recognizer directly to extended grammars, a few of the foregoing definitions have to be revised.

As before, a grammar rule is written $A \rightarrow \alpha$, but with α now a regular expression with N_α symbols (elements of V). Defining $T_\alpha^+ = 1...N_\alpha$ and $T_\alpha = 0...N_\alpha$, regular expression α can be characterized by

1. a mapping $\phi_\alpha : T_\alpha^+ \rightarrow V$ associating a grammar symbol to each number.

2. a function $succ_\alpha : T_\alpha \rightarrow 2^{T_\alpha^+}$ mapping each number to its set of successors. The regular expression can start with the symbols corresponding to the numbers in $succ_\alpha(0)$.

3. a set $\sigma_\alpha \in 2^{T_\alpha}$ of numbers of symbols the regular expression can end with.

Note that 0 is not associated to a symbol in V and is not a possible element of $succ_\alpha(k)$. It can be element of σ_α though, in which case there is an empty path through the regular expression.

We define an item as a pair $(A \rightarrow \alpha, k)$, with the interpretation that number k is 'just before the dot'. The correspondence with dotted rules is the following. Let $\alpha = B_1...B_l$, then α is a simple regular expression characterized by $\phi_\alpha(k) = B_k$, $succ_\alpha(k) = \{k+1\}$ if $0 \leq k < l$, $succ_\alpha(l) = \emptyset$, and $\sigma_\alpha = \{l\}$. Item $(A \rightarrow \alpha, 0)$ corresponds to the initial item $A \rightarrow .\alpha$ and $(A \rightarrow \alpha, k)$ to the dotted-rule item with the dot just after B_k.

The predicate *final* for the new kind of items is defined by

$final((A \rightarrow \alpha, k)) = (k \in \sigma_\alpha)$

Given a set q of items, we define

$$\begin{aligned}ini(q) = \{(A \rightarrow \alpha, 0) | (B \rightarrow \beta, l) \in q \wedge \\ k \in succ_\beta(l) \wedge \phi_\beta(k) \Rightarrow^* A\gamma\}\end{aligned}$$

The function *pop* becomes set-valued and the transition function can be defined in terms of it (remember: $\overline{q} = q \cup ini(q)$):

$$pop((A \rightarrow \alpha, l)) = \{(A \rightarrow \alpha, k) | l \in succ_\alpha(k)\}$$

$$\begin{aligned}goto(q, B) = \{(A \rightarrow \alpha, k) | \phi_\alpha(k) = B \wedge I \in \overline{q} \wedge \\ I \in pop((A \rightarrow \alpha, k))\}\end{aligned}$$

A recursive ascent recognizer is now implemented by

$$\begin{aligned}[q](i) = \overline{[q]}(x_{i+1}, i+1) \cup \\ \{(I, j) | J \in ini(q) \wedge final(J) \wedge \\ (I, j) \in \overline{[q]}(lhs(J), i)\} \cup \\ \{(I, i) | I \in q \wedge final(I)\}\end{aligned}$$

$$\begin{aligned}\overline{[q]}(B, i) = \{J, j) | J \in q \wedge J \in pop(I) \wedge \\ (I, j) \in [goto(q, B)](i)\} \cup \\ \{(I, j) | (J, k) \in [goto(q, B)](i) \wedge K \in ini(q) \wedge \\ K \in pop(J) \wedge (I, j) \in \overline{[q]}(lhs(J), k)\}\end{aligned}$$

The initial state q_0 is $\{(S' \rightarrow S, 0)\}$, and a sentence $x_1 ... x_n$ is grammatical if $((S' \rightarrow S, 0), n) \in [q_0](0)$. The recognizer is deterministic if

1. there is no shift-reduce or reduce-reduce conflict, i.e. every state has at most one final item, and in case it has a final item it has no items $(A \rightarrow \alpha, j)$ with $k \in succ_\alpha(j) \wedge \phi_\alpha(k) \in V_T$.
2. for all reachable states q, $q \cap ini(q) = \emptyset$, and for all I there is at most one $J \in \overline{q}$ such that $J \in pop(I)$.

In the deterministic case, the analysis of section 4 can be repeated with one exception: extended grammar items can not be represented by a non-terminal and an integer that equals the number of symbols before the dot, as this notion is irrelevant in the case of regular expressions. In standard presentations of deterministic LR-parsing this leads to almost unsurmountable problems [5].

8 Conclusions

We established a very simple and elegant implementation of LR(0) parsing. It is easily extended to LALR(k) parsing by letting the functions $[q]$ produce pairs with final items only after inspection of the next k input symbols.

The functional LR-parser provides a high-level view of LR-parsing, compared to conventional implementations. A case in point is the ubiquitous stack, that simply corresponds to the procedure stack in the functional case. As the proof of a functional LR-parser is not hindered by unnecessary implementation details, it can be very compact. Nevertheless, the functional implementation is as efficient as conventional ones. Also, the notion of memo-functions is an important primitive for presenting algorithms at a level of abstraction that can not be achieved without them, as is exemplified by this paper's presentation of both the recognizers and the parse forests.

For non-LR grammars, there is no reason to use the complicated Tomita algorithm. If indeed non-deterministic LR-parsers beat the Earley algorithm for some natural language grammars, as claimed in [4], this is because the number of LR(0) states may be smaller than the size of I_G for such grammars. Evidently, for the grammars examined in [4] this advantage compensates the loss of efficiency caused by the non-polynomiality of Tomita's algorithm. The present algorithm seems to have the possible advantage of Tomita's parser, while being polynomial.

Acknowledgement

A considerable part of this research was done in collaboration with Lex Augusteyn and Frans Kruseman Aretz. Both are colleagues at Philips Research.

References

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

2 G.H. Roberts, Recursive Ascent: An LR Analog to Recursive Descent, *SIGPLAN Notices* (1988) 23(8):23-29.

3 J. Hughes, *Lazy Memo-Functions* in *Functional Programming Languages and Computer Architecture* edited by J.-P. Jouannaud, *Springer Lecture Notes in Computer Science* (1985) 201.

4 M. Tomita, *Efficient Parsing for Natural Language* (Kluwer Academic Publishers, 1986).

5 P.W. Purdom and C.A. Brown, Parsing extended LR(k) grammars, *Acta Informatica* (1981) 15:115-127.

6 A.V. Aho and J D. Ullman, *Principles of Compiler Design* (Addison-Wesley publishing company,1977)

7 A.V. Aho and J.D. Ullman, *The theory of parsing, translation, and compiling* (Prentice Hall Inc. Englewood Cliffs N.J.,1972).

8 B.A. Sheil. *Observations on Context Free Parsing* in *Statistical Methods in Linguistics* (Stockholm (Sweden) 1976).
Also: Technical Report TR 12-76, Center for Research in Computing Technology, Aiken Computation Laboratory, Harvard Univ., Cambridge (Massachusetts).

9 J. Earley, 1970. An Efficient Context-Free Parsing Algorithm, *Communications ACM* 13(2):94-102.