KYBERNETIKA — VOLUME *33* (1997), NUMBER 3, PAGES 311–332

CONSTRUCTION OF A ROBUST PARSER FROM A DETERMINISTIC REDUCED PARSER

MARTIN PLÁTEK

A formal method for the automatic construction of an error recovery part for a given parser is presented. We use a new formal model of parser, and a new notion of headsymbol instead of skeletal symbols. We guarantee in a formal way the recovery from any type of errors and we separate the (in)correct input parts with the help of the hierarchized output.

1. INTRODUCTION

The starting point of the present study is [1]. The great advantage of [1] is the exact formulation of guarantees for the error recovery in the terms of formal languages and automata. We try to work in a similar way. We use *headsymbols* instead of *skeletal symbols*, and we present push-down automata as special list automata.

This model of automata allows to introduce moving-trees. Any computation of such an automaton can be expressed as a moving-tree. We consider moving-trees as the output of the automata.

In [1] the recovery from *"skeletal errors"* is left open. Our method guarantees a recovery from any type of errors (full robustness).

There are no problems to determine the set of headsymbols for a given parser. The same problem for skeletal sets is much more complex, maybe unsolvable (see [4]).

Further motivation to use headsymbols is the observation, that some popular programing languages like Pascal do not contain suitable skeletal sets, but their parsers ordinarily work with a suitable set of headsymbols.

As headsymbols, for example, some of the key words of Pascal: **begin, if, repeat, until**, etc. can be used.

In [1] the given guarantees are conditioned by the property of error sensitivity of the parsers (a parser operates only inside of the correct prefixes of the input text). Instead of error sensitive parsers we work with reduced parsers (similar concept as for grammars). For reducing of parsers there are much more effective algorithms than for the transformation in the error sensitive parsers (see [3]). The reduction of a parser surely does not increase its size. That is not the case if we consider the transformation in the error sensitive parser.

Our model of automata can serve as a formal device for the interpretation of the standard parsing methods ($LL(1), LR(1), \ldots$). The tables describing such parsers can be transformed into sets of instructions for the list automata in a simple algorithmic way. This makes our method independent on the used method of parsing.

The remainder of the paper is divided in seven sections:

2. Basic notions
3. Properties of parsers
4. Headsymbols
5. A construction of an error recovery part to a reduced parser
6. Formal properties of M_R
7. Examples
8. Concluding remarks

The main concept is introduced in Section 4. The construction of the robust parser is presented in Section 5, and the formal properties of this construction are formulated in Section 6. We hope that the titles of the other sections summarize sufficiently theirs content.

2. BASIC NOTIONS

We work with automata of the following form: The memory is a linear doubly linked list with a left-hand side and a right-hand side sentinel. Every item in the list can store one symbol. The first item always contains the symbol # (left-hand side sentinel), the last one the symbol \$ (right-hand side sentinel). The automaton has a workhead which can read the content of the visited item and the content of its right-hand side neighbour (cf. Figure 1.1). The automaton can perform the following basic operations: *MVR* – a move to the right, *DEL* – a deletion of the visited item and a move to the left. *IN(symbol)* – insertion of the *symbol* immediately to the right of the visited item and a move on the inserted item.

Definition 2.1. A list automaton (L–automaton) M is a 5-tuple $M = (Q, A, B, It, q_0)$, where Q is a finite set (of states), A is a finite alphabet (input alphabet), B is a finite alphabet (working alphabet), where $B \supseteq (A \cup \{\#, \$\})$, and $q_0 \in Q$ (initial and accepting state).

The elements of the finite set It are called instructions and have the following form: $[q_1, b, a] \rightarrow [q_2, op]$, where $q_1, q_2 \in Q$, $b \in B$, $a \in (A \cup \$)$; $op \in \{MVR, DEL, IN(b)$, for $b \in B - \{\#, \$\}\}$.

M is deterministic if the set It does not contain two different instructions of the form $[q, b, c] \rightarrow [q_2, op_1]$, $[q, b, c] \rightarrow [q_1, op]$ (with the same left-hand side and different right-hand side).

In the present paper only deterministic automata are considered.

We represent a configuration of M via a triple $K = (wb, q, v)$, where w represents the string of symbols in the list on the left-hand side of the symbol visited by the

head (left-hand side of K), b means the symbol visited by the head, v is the string on the nonvisited part of the list (right-hand side of K) and q means the current state of M. The string wb may be considered as the content of the pushdown of M (b is the top).

The number of symbols in the word wb is called the size of the left-hand side of K (we denote it as $szl(K)$), and the number of symbols in the word v is called the size of the right-hand side of K (we denote it as $szr(K)$).

Let $K_1 = (wb, q, cv)$ be a configuration of M, $i = [q, b, c] \rightarrow [q_1, op]$ some instruction of M.

We write $K_1 \Rightarrow_i K_2$ in the following cases:

(a) $op = MVR$, $K_2 = (wbc, q_1, v)$

(b) $op = DEL$, $K_2 = (w, q_1, cv)$

(c) $op = IN(d)$, $K_2 = (wbd, q_1, cv)$.

The notation $K_1 \Rightarrow K_2$ means, that there is an instruction i such that $K_1 \Rightarrow_i K_2$. We say that $K_1 \Rightarrow K_2$ is a *step* from K_1 to K_2. The reflexive and transitive closure of the relation $\Rightarrow$ is denoted by $\Rightarrow^*$.

Configuration of the form $(\#, q_0, v\$)$, where $v \in A^*$, is called a *starting* configuration of M.

Let λ denotes the empty word. A configuration of the form $(\lambda, q_0, \$)$ is an *accepting* configuration (the list contains only a sentinel, and the head of M is not positioned on the list).

Any configuration K such that there exists no step of the form $K \Rightarrow K_1$ is called a *halting* configuration.

We say that any sequence C of steps $K_1 \Rightarrow K_2, K_2 \Rightarrow K_3, \ldots, K_{n-1} \Rightarrow K_n, \ldots$ is a *computation* of M.

If K_1 is a starting configuration, we call C a *starting* computation. If K_n is a halting configuration, we call C a *halting* computation.

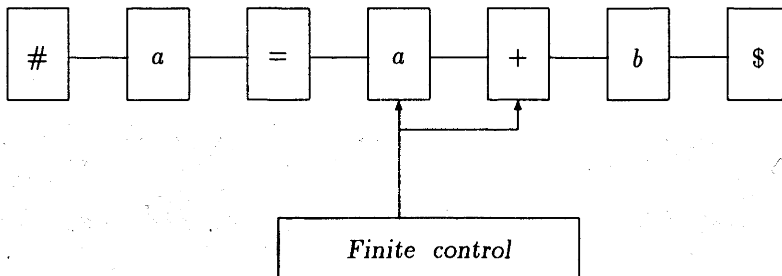

Fig. 1. **L-automaton.**

We say that a computation is *completed* if there does not exist a step prolonging this computation.

There are two types of completed computations: infinite and halting computations.

Any halting configuration which is in the form $(\#v, q, u\$)$ is called an *error* configuration.

We denote $L(M) = \{v \in A^* \mid (\#, q_0, v\$) \Rightarrow^* (\lambda, q_0, \$)\}$.

We give two examples of list automata.

Example 2.2. Automata M_A and M_C recognize the same language: set of words of the form $a_0 = a_1 + \ldots + a_n$, where $a_i \in \{a, b\}$.

The automaton M_A proceedes in the following way: At first without any checking moves to the right-hand side sentinel. All the checking is performed by the deleting with the outlook on the right-hand side sentinel.

$M_A = (\{q_0, q_1, q_a, q_=, q_\#\}, \{a, b, +, =\}, \{a, b, +, =, \#, \$\}, ITA, q_0)$, ITA (set of instruction) is described with the help of the following schemes (metarules):

(1) $(\{\#, a, b, +, =\}, q_0, \{a, b, +, =\}) \rightarrow (q_0, MVR)$
(2) $(\{a, b\}, q_0, \{\$\}) \rightarrow (q_1, DEL)$
(3) $(\{+\}, q_1, \{\$\}) \rightarrow (q_a, DEL)$
(4) $(\{a, b\}, q_a, \{\$\}) \rightarrow (q_1, DEL)$
(5) $(\{=\}, q_1, \{\$\}) \rightarrow (q_=, DEL)$
(6) $(\{a, b\}, q_=, \{\$\}) \rightarrow (q_\#, DEL)$
(7) $(\{\#\}, q_\#, \{\$\}) \rightarrow (q_0, DEL)$;

A scheme $(P, p, R) \rightarrow (q, op)$ represents a set of instructions of the form $[p, a, b] \rightarrow [q, op]$, where a is from P and b is from R, e.g. the scheme (2) represents two instructions: $[q_0, a, \$] \rightarrow [q_1, DEL]$ and $[q_0, b, \$] \rightarrow [q_1, DEL]$. Similarly we can use the sets of states instead of single states.

The automaton M_C computes in a different way. It inserts the symbol C instead of the symbol $=$. All the visited symbols on the righthand side of C are immediatly deleted.

$M_C = (\{q_0, q_r, q_a, q, q_=, q_+, q_1, q_2, q_\$\}, \{a, b, +, =\}, W, ITC, q_0)$, where $W = \{a, b, +, =, C, \#, \$\}$, and ITC is a set of instructions described with the help of following schemes:

(1) $(\{\#\}, q_0, \{a, b\}) \rightarrow (q_r, MVR)$
(2) $(\{a, b\}, q_r, \{=\}) \rightarrow (q_=, MVR)$
(3) $(\{=\}, q_=, \{a, b\}) \rightarrow (q, DEL)$
(4) $(\{a, b\}, q, \{a, b\}) \rightarrow (q_=, IN(C))$
(5) $(\{C\}, q_=, \{a, b\}) \rightarrow (q_a, MVR)$
(6) $(\{a, b\}, q_a, \{+\}) \rightarrow (q_+, DEL)$
(7) $(\{a, b\}, q_a, \{\$\}) \rightarrow (q_a, DEL)$;
(8) $(\{C\}, q_+, \{+\}) \rightarrow (q_+, MVR)$
(9) $(\{+\}, q_+, \{a, b\}) \rightarrow (q_2, DEL)$
(10) $(\{C\}, q_2, \{a, b\}) \rightarrow (q_a, MVR)$
(11) $(\{C\}, q_a, \{\$\}) \rightarrow (q_1, DEL)$
(12) $(\{a, b\}, q_1, \{\$\}) \rightarrow (q_\$, DEL)$
(13) $(\{\#\}, q_\$, \{\$\}) \rightarrow (q_0, DEL)$

We introduce the concept of *string-tree* in order to prepare the introduction of *moving-trees*. Moving-trees will describe the structure of the computations of the list automata, and we will consider moving-trees as parsing structures of the corresponding input strings.

Definition 2.3. A *string-tree* S over an alphabet B is a rooted tree $S = (U, E)$, where U (set of nodes) is a set of indexed symbols, where the symbols are from B,

and the indexes are integers (formally $U \subset B \times Int$). U is ordered by the indexes of its members. The ordering of U (signed by $>$) has the following properties:

[1] (left-to-right pathes) If a path leads from a_i to another node b_j, then $a_i < b_j$.

[2] (projectivity) If u_1, u_2, u_3 are nodes of S, $u_1 < u_2 < u_3$, $[u_1, u_3]$ is an edge from E, then there is a path leading from u_1 to u_2.

We can see, that the root is always the first node by $<$.

Let $S = (U, E)$ be a string-tree over B, $U = \{u_1, \ldots, u_n\}$, $u_i = a_j$, where $a \in B$. We write $Sym(u_i) = a$. We denote $Str(S) = Sym(u_1) \ldots Sym(u_n)$. By $Str(S, A)$ we denote a string which we get from $Str(S)$ by removing all its symbols not belonging to the alphabet A.

We say that the string-trees $S_1 = (U_1, E_1)$ and $S_2 = (U_2, E_2)$ are isomorphic if there is a one-to-one mapping f from U_1 to U_2 such that a/b is from E_1 iff $f(a)/f(b)$ is from E_2, and $a < b$ iff $f(a) < f(b)$, and $Str(S_1) = Str(S_2)$.

We introduce string-trees and their isomorphism since we will not make any difference between two isomorphic moving-trees.

Let u_1, u_2 be nodes of some string-tree S. We say that u_2 depends on u_1 if a path of S leads from u_1 to u_2. We denote an edge $[u_1, u_2]$ usually as u_1/u_2 and a *bundle* of edges $u_1/u_2, \ldots, u_1/u_n$ as $u_1/(u_2, \ldots, u_n)$. We can describe any string-tree by nesting of the previous description of bundles and we get the so called */-description*.

Let S be a string-tree. We write $S_1 \ll S$ if S_1 is a string-tree such that it is a subtree of S, and any node of S depending on the root of S_1 is from S_1.

Let S be a string-tree. We write $S_1 \prec S$ if S_1 is a subtree of S such that $Str(S_1)$ is a substring of $Str(S)$.

We can see that if $S_1 \ll S$ then $S_1 \prec S$.

Let be $S_1 \prec S$. We denote by $S - S_1$ a string-tree which contains all nodes from S not depending on any node from S_1.

Example 2.4. Let $S_1 = (U_1, E_1)$, where

$U_1 = \{\#_0, a_1, =_2, a_3, +_4, b_5\}$, $\qquad E_1 = \{\#_0/a_1, a_1/=_2, =_2/a_3, a_3/+_4, +_4/b_5\}$.

We get the /-description of S_1:

$$\#_0/a_1/=_2/a_3/+_4/b_5.$$

Example 2.5. Let $S_2 = (U_2, E_2)$, where

$U_2 = \{\#_0, a_1, =_3, C_4, a_5, +_6, b_8\}$, $\qquad E_2 = \{\#_0/a_1, a_1/(=_3, C_4), C_4/(a_5, +_6, b_8)\}$.

We get the /-description for S_2:

$$\#_0/a_1/(=_3, C_4/(a_5, +_6, b_8)).$$

Figure 2 represents S_2.

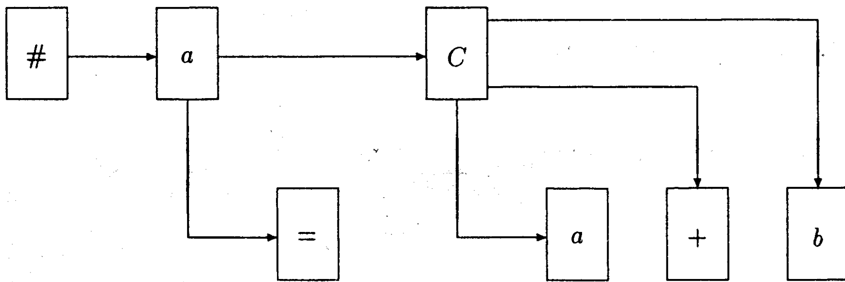

Fig. 2. String-tree S_2.

Definition 2.6. Let $C = K_1 \Rightarrow_{I_1} K_2, K_2 \Rightarrow_{I_2} K_3, \dots, K_n \Rightarrow_{I_n} K_{n+1}, \dots$ be a computation of M. We can see that I_j means the instruction used in the jth step of C. We call the sequence of instructions $I_1, \dots, I_n, \dots$ *trace* of the computation C, and denote it by $TR(C)$.

We will define the moving-tree S by C (or by $TR(C)$). Roughly speaking the nodes of S correspond to the visited items during C and the edges correspond to the actions of C causing the first visits on the single items. Let us recall that computations in a general case can start on an item placed inside of the list. This fact slightly complicates the next definition.

First we outline the meaning of the symbols (variables) used in the step-by-step definition of the moving-tree.

Scan	– the position of the head	– [integer]
Instr	– the order of the current instruction	– [natural number]
U	– the current set (of indexes) of nodes	– [set of integers]
E	– the current set of edges	– [set of edges]
Act	– the indices of nodes currently in the list	– [set of integers]
Dep	– a node waiting for its tree-predecessor	– [empty or one-node set]

Let C be a computation of $M = (Q, A, B, It, q_0)$, and $TR(C) = I_1, \dots, I_n, \dots$. *Moving-tree* $S = (U, E)$ by C (or by $TR(C)$) is a string-tree over B, defined step by step, using in the ith step the information corresponding to the prefix of $TR(C)$ $I_1, \dots, I_i$:

1. **Initialization.** Let us suppose that I_1 has the form $[q_1, b, a] \to [q_2, op]$.

   We consider the following cases:

   If $op = DEL$

   then we set $Scan := -1$, $Instr := 1, U := \{b_0\}$, $Dep := \{b_0\}$, $E := \{empty\ set\}$, and $Act := \{empty\ set\}$.

   If $op = MVR$

   then we set $Scan := 1$, $Instr := 1$, $U := \{b_0, a_1\}$, $E := \{b_0/a_1\}$, $Act := \{0, 1\}$, and $Dep := \{empty\ set\}$.

If $op = IN(c)$

then we set $Scan := 1$, $Instr := 1$, and $U := \{b_0, c_1\}$, $E := \{b_0/c_1\}$, $Act := \{0, 1\}$, and $Dep := \{empty\ set\}$.

2. **Inductive step.** Let us suppose $Scan = i$, $Instr = k$, $I_{k+1} = [q_{k+1}, a, d] \rightarrow [q_{k+2}, op_{k+1}]$, and U is nonempty.

If the node with the index i is not contained in U it is supposed that $i \notin Act$ and Dep is not empty;

then we set at first $U := U \cup \{a_i,\}$, $Act := Act \cup \{i\}$, and $E := E \cup \{a_i/u\}$, where u is the node from Dep. After that we set $Dep := \{empty\ set\}$.

Further we consider the following cases:

If $op_{k+1} = DEL$, and there is in Act a number lower than i, and p is the greatest number from Act, less than i,

then we set $Scan := p$, $Act := Act - \{i\}$, $Instr := k + 1$.

If $op_{k+1} = DEL$, and there is not in Act any number lower than i, and n is the smallest index of the nodes from U

then we set $Scan := n - 1$, $Instr := k + 1$, $Act := Act - \{i\}$, and $Dep := \{a_i\}$.

If $op_{k+1} = MVR$, and m is the maximum of indices of the nodes from U

then we set $Scan := m + 1$, $Instr := k + 1$, $Act := Act \cup \{m + 1\}$, $U := U \cup \{d_{m+1}\}$, and $E := E \cup \{a_i/d_{m+1}\}$.

If $op_{k+1} = IN(c)$, and m is the maximum of indices of the nodes from U

then we set $Scan := m + 1$, $Instr := k + 1$, $Act := Act \cup \{m + 1\}$, $U := U \cup \{c_{m+1}\}$, and $E := E \cup \{a_i/c_{m+1}\}$.

We say that a string-tree S is a *moving-tree* by M if there is a computation C by M, such that S is a moving tree by C. Let a_i/b_j be an edge of a moving-tree S by the computation C. We will define the *history* of a_i/b_j. If the a_i, respectively b_j, is entered only once during C, the history of a_i/b_j is the single instruction realizing the first visit on a_i (deleting instruction), or on b_j (nondeleting instruction). In the other case the history (so called complete history) of a_i/b_j, is an ordered pair of instructions (I_k, I_l), where I_k performes the first visit on b_j, and I_l performes the deletion of b_j, followed by the visit of a_i.

The set of all moving-trees by M is denoted as $S(M)$. The subset of $S(M)$ of all moving-trees created by an accepting computation of M is denoted as $S_A(M)$. The subset of $S(M)$ of all moving-trees created by a starting computation of M is denoted as $S_S(M)$. A moving-tree S from $S(M)$ is called *deleting-tree* by M if the histories of all edges of S are complete.

Example 2.7. S_1 from the Example 2.4 is a member of $S_A(M_A)$, and S_2 from the Example 2.5 is from $S_A(M_C)$. M_A and M_C are defined in the Example 2.2.

3. PROPERTIES OF PARSERS

Convention. M denotes a deterministic list automaton $M = (Q, A, B, It, q_0)$.

Definition 3.1. If M does not insert symbols from the input alphabet A then M is called a *parser*. We say that a parser M is *reduced* if any instruction of M can be used in some accepting computation of M.

Let us recall that $szl(K)$ means the size of the left-hand side of the configuration K, and $szr(K)$ the size of the right-hand side of the configuration K. We call the expression $szl(K) + szr(K)$ the *size of* K. Let us note that we use the sign $\times$ for multiplication. The next theorem shows that parsers are computing in the linear time, and we will notice a useful consequence of this fact. We will take advantage of this consequence by the construction of a robust parser.

Theorem 3.2. Let M be a reduced parser. Then a constant r exists such that the two following assertions hold:

(a) The number of steps in a computation C of M beginning with a configuration of the size l is smaller than $l \times r$.

(b) Let C_m be a completed computation of M, where K_f is the first configuration of C_m, and K_l the last one. Then it holds:

$$r \times (szr(K_f) - szr(K_l)) > szl(K_l) - szl(K_f).$$

Proof. First we prove the assertion (a). We divide the proof into four observations:

- **Observation 1.** Let C be an infinite computation of M. Then there is an infinite subcomputation C_1 of C such that the item scanned in the first configuration of C_1 is not deleted during C_1, and the operation *MVR* is not performed during C_1 at all.

  Proof. If we suppose the opposite, one of the two following cases occurs:

  – in C_1 the operation *MVR* is used infinitely times

  or

  – in C_1 an unlimited number of items not inserted during C_1 is deleted.

  But that is in a contradiction with the finite size of the first configuration of C_1.

- **Observation 2.** Let us suppose that there is an infinite computation of M. Then M cannot be a reduced parser.

  Proof. Let C be an infinite computation of M. Let S be the moving-tree by C. Let C_1 be the maximal subcomputation of C, where in the trace of C_1 there is not any instruction performing the operation *MVR*, and all deleted items by

C_1 are inserted during C_1. By the Observation 1 C_1 is infinite. We take the first instruction I, performed by C_1. We can see that instruction I performs the operation insert. Any computation which used I is not a halting computation because of the determinism of M. Therefore an accepting computation cannot use the instruction I. □

- **Observation 3.** Let m be the number of instructions of M, and C_2 be a computation of M with a moving-tree S_2 with the following property: In the histories of edges of S_2 there is not any instruction performing the operation *MVR*, and the number k of edges of S_2 is greater than n^n, where $n = m^2$. Then there is an infinite computation by M.

  Proof. By the computation C_2 only the items inserted by C_2 are deleted. The number k is big enough to ensure that one of the two following cases occurs:

  (i) some two edges in a path of S_2 have the same history, or

  (ii) some two edges of S_2 with a common node have the same history.

  That means, by determinism of M, that an endless cycle begins in C_2 which infinitely times inserts (and possibly deletes some of the inserted items again). □

- **Observation 4.** Let M be a reduced parser and m be the number of instructions of M. Let $r = 2 \times n^n + 1$, where $n = m^2$. Then in any computation C of M longer than r steps the operation *MVR* or a deletion of the starting item of C, is performed at least once.

  Proof. If we suppose the opposite, then there is a moving-tree by M fulfilling the assumptions of the Observation 3. By the Observation 2 the automaton M cannot be a reduced automaton. □

Now we can finish the proof of the assertion a). We show how it follows from the Observation 4. Let C be a computation by M. We can divide C in such subcomputations $C_1, C_2, \ldots, C_j$, where in the first configuration of any C_i the scanned item is visited during C for the first time (after *MVR* or *DEL*). From the Observations 4 it follows that the length of such computations can be choosen in such a way that it does not exceed the number r.

The proof of the assertion b): From a) it follows that C_m is a finite computation. C_m is a completed computation, therefore the configuration K_l is a halting configuration. We divide the remaining proof in two cases:

Let K_l be an accepting configuration. Then the $szl(K_l) = 0$, and $szr(K_l) = 1$. We can see that there is not such K_f that $K_f \Rightarrow K_l$, and $szl(K_f) + r \times szr(K_f) \leq r$, since the size of K_f is at least 2.

Let us suppose that K_l is an error configuration. In this case the assertion b) follows from the proof of the assertion a). We can see that the left-hand side of the list can be prolonged at most by $r-1$ inserted symbols between any two *MVR*-steps during C_M. From this observation the assertion b) follows directly. □

Remark. In [6] it is shown that any parser M can be automatically transformed to a reduced parser M_r such that $S_A(M) = S_A(M_r)$. An efficient algorithm for the reduction of parsers is presented in [3].

We use the properties of reduced parsers to formulate an error recovery technique in the next section.

Theorem 3.3. The parsers recognize exactly the class of deterministic context-free languages (DCFL).

Proof. We divide the proof in two cases:

a) Any language recognized by a parser is in $DCFL$:
A parser M, which recognizes $L(M)$ can be simulated step by step by a deterministic pushdown automaton M_1 so that the content of the left-hand side part of the considered configuration of M is kept in the pushdown store of M_1.

b) Any language from $DCFL$ is recognized by some parser:
From [7] it follows that the class of languages recognized by deterministic list automata without look ahead, using only the operations *MVR*, *DEL* and rewriting of symbols, is equal to the class $DCFL$. We see that a list automaton M, which uses *MVR*, *DEL* and rewriting only, can be easily simulated by a parser M_1, simulating M in such a way that instead rewriting by a symbol b, it uses a pair of operations *DEL* and $IN(b)$. □

Remark. We have not yet explicitly introduced an output of the parser. In the sequel we consider as the output to a given computation the responding moving-tree. We demonstrate then that moving-trees on the output can serve at least so well as the output tape.

Let us consider a parser M_0 with an additional output tape. Let $M_0 = (Q, A, B, C, It_0, q_0)$, where for the simplicity C and B are disjoint, the instructions of IT_0 have the form

$$[q, a, b] \rightarrow [p, op, c],$$

where c (output symbol) can be a symbol from C, or λ (empty string). The instructions are performed in an obvious way as by usual parsers, enhanced by the sequential writing of the output symbols on the output tape.

It is easy to construct a parser M_1, which simulates M_0 in such a way that the actual simulated situation on the output tape of M_0 can be derived from the related moving-tree of M_1.

The construction of M_1 can be done in the following way: Let

$$M_1 = (Q \cup Q_O, A, B \cup C, IT_1, q_0),$$

where

$$Q_O = \{(q, i); q \in Q, i \in \{1, 2\}\},$$

and for any instruction from IT_0 of the form $[q,a,b] \rightarrow [p,op,c]$ with $c \in C$, we put in IT_1 the following triple of instructions:

$$[q,a,b] \rightarrow [(p,1), IN(c)],$$
$$[(p,1),c,b] \rightarrow [(p,2), DEL],$$
$$[(p,2),a,b] \rightarrow [p,op].$$

If c is an empty string, we put the single instruction $[q,a,b] \rightarrow [p,op]$ in IT_1.

We can see from the previous construction that the moving-tree to an accepting computation of M_1 contains simultaneously the information about the input word and about the corresponding output word computed by M_0. That observation allows without loss of generality to use moving-trees instead of the usual sequential output tape.

4. HEADSYMBOLS

We introduce *headsymbols* in this section. The headsymbol can be an input symbol as well an inserted symbol. Our recovery method is based on the properties of headsymbols, namely of input-headsymbols. Programming languages use many input symbols with the properties of headsymbols. For instance in a parser of Pascal the keywords **begin**, **if**, **while**, and others can usually be considered as headsymbols. The symbol **case** is not suitable for that purpose, because it has two different meanings in Pascal.

Definition 4.1. Let $M = (Q, A, B, It, q_0)$, $c \in B$. We denote

$$SV(c) = \{q \in Q;\ \text{there is some } [q_1, a, c] \rightarrow [q, MVR] \in IT, \text{ or some } [q_3, b, d] \rightarrow [q, IN(c)] \in IT\}.$$

We denote

$$SE(c) = \{q \in Q;\ \text{there is some } [q_1, c, d] \rightarrow [q, DEL] \in IT\}.$$

We call c from B a *headsymbol* of M if

$$card\,(SE(c)) = 1, \text{ and } card\,(SV(c)) \leq 1.$$

Remark. In any computation of a reduced parser M the state of the first visit on a headsymbol c is unambiguously determined, and the state immediately after deleting c is unambiguously determined, too. For a reduced parser M, where $L(M)$ is nonempty, the left-hand side sentinel is a headsymbol and the right-hand side sentinel is not a headsymbol.

Example 4.2. The set $\{\#, =, +\}$ is the set of headsymbols of the automaton M_A from the Example 2.2, and the set $\{\#, =, C, +\}$ is the set of headsymbols of the automaton M_C.

5. A CONSTRUCTION OF AN ERROR RECOVERY PART TO A REDUCED PARSER

For any reduced parser M we construct an automaton M_1 realizing an error recovery part to M. M_R will be a robust automaton composed from M and M_1. The purpose of any computation of M_1 is to restart a computation of M (a recovery), to assure, that the state and place of the restart cannot be inproperly chosen.

Conventions.

Let $b \in B$. $DP(M,b) = \{c \in A;$ there is $T \in S_A(M)$ with a node containing the symbol c depending on some node containing the symbol $b\}$.

We can see that $DP(M,\#) = A$ (A is the input alphabet of M).

RB stands for a set of input symbols containing the right-hand side sentinel, LB stands for some set of headsymbols containing the left-hand side sentinel. LB and RB are sets of boundary symbols (left-hand side and right-hand side).

$MRB \subseteq \{(a,b); a \in LB, b \in RB,$ and b is a headsymbol such that there is a $[q,a,b] \to [SV(b), MVR] \in IT\}$.

$DLL \subseteq \{(a,b); a \in LB, b \in RB$ such that there is a $[q,a,b] \to [SE(a), DEL] \in IT\} \cup \{(\#,\$)\}$. Moreover DLL always contains $(\#,\$)$.

M_R works in the following way:

(a) Any starting computation of M_R begins with some starting configuration of M. M computes until the first halting configuration K of M.

(b) In the case that K is an error configuration of M, a nonempty computation of M_1 from K is started.

(c) Any completed computation of M_1 finishes by a configuration of M. If it is an error configuration M_1 is started again. If it is an accepting configuration, M_R accepts it. In other cases a nonempty computation of M is started.

We can see that M_R halts always in an accepting configuration of M. At the first glance we see that the parsing of correct words from L(M) by M_R, is not burdened by the error recovery component (M_1).

It remains to describe the recovery automaton M_1.

The description of M_1.

M_1 inserts two types of symbols neither of which is contained in B:

a) error signs – serve as error messages.

b) special signs – serve to separate the output components, computed by M from the components computed by M_1.

M_1 consists of the following eight commands, which should be performed in the outlined order:

(1) M_1 starts always by this command. M_1 inserts an error sign if the visited symbol is not a special sign.

(2) The head of M_R moves step by step to the right and stops when looking ahead a symbol from RB. We denote the look-ahead symbol as b (visited is the left-hand side neighbour of b).

(3) M_1 performs the operation DEL (deleting with a move to the left) until an error sign or a special sign is encountered. Then M_1 prolongs the deletion until a symbol from LB is found. We denote this symbol as a.

(4) If (a,b) is from MRB, the head inserts a special sign between a and b and erases it again. Then M_1 moves the head on b, and M_R is transfered to the state $SV(b)$ of M.

(5) If (a, b) is not from MRB, but b is a headsymbol from $DP(M, a)$, the head inserts a special sign between a and b, moves to b, and M_R is transfered to the state $SV(b)$.

(6) If b is not a headsymbol and if b is from $DP(M, a)$ a special sign is inserted between a and b. After that the head moves to the right until a symbol from RB is in the look-ahead position. The symbol in the look-ahead position is denoted as b. After that M_1 deletes items until the symbol marked as a is visited (the special symbol is erased), and continues with the command (4).

(7) If b is not from $DP(M, a)$, and (a, b) is from DLL, M_1 inserts a special sign between a and b, and erases it again. Then the headsymbol a is deleted, and M_R is transfered to the state $SE(a)$.

(8) If the previous conditions are not fulfilled, M_1 deletes a, and continues in deleting until a symbol from LB is reached. The visited symbol is denoted as a. The computation of M_1 continues with the command (4).

6. FORMAL PROPERTIES OF M_R

We suppose in the following text that M_R is constructed by the previous construction (Section 5) from a reduced parser denoted as M. The properties of M_R are formulated as theorems, the properties of M and its headsymbols are formulated as claims.

Theorem 6.1. The two following assertions hold:

(a) M_R is a parser and any starting completed computation of M_R is an accepting computation (the robustness of M_R).

(b) $S \in S_A(M_R)$ is a member of $S_A(M)$ if and only if S does not contain any error sign.

Proof.

ad (a): We can see that the commands $(1), \ldots, (8)$ from the description of M_1 can be carried out using only the operations MVR, DEL and $IN(a)$, where a is not from A. Therefore we can construct M_R as a parser.

We know that any starting completed halting computation of M_R ends by an accepting configuration of M. We need only to show that any starting completed computation of M_R halts. Let C be a starting completed computation of M_R. C following the construction of M_R, and Theorem 3.2 is divided in halting computations of M and halting computations of M_1. We need to show that there is only a limited number of switches between computations of M and computations of M_1.

We will show that the number of switches between computations of M and computations of M_1 during C is less than

$$f(K_1) = szl(K_1) + r \times szr(K_1),$$

where K_1 denotes the starting configuration of C, and r is the constant given by Theorem 3.2.

We need only to show, that the function

$$f(K) = szl(K) + r \times szr(K)$$

is decreasing during C, if we check only the values of f of the first configuration of C and of any switching configuration between computations of M and computations of M_1 and vice versa.

We first consider a completed subcomputation C_1 of C by M_1. After the use of a command of the type (4),(5) or (6), the szr decreases and the szl does not increase, after a use of the commands (7) or (8) the szl decreases and the szr does not increase. Therefore the value of f of the last configuration of C_1 is lower than the value of the first configuration of C_1. The first and the last configuration of C_1 are switching configurations.

To complete the proof we consider a subcomputation C_M of C by M. Let K_f be the switching configuration and the first configuration of C_M, K_l the last configuration of C_M and simultaneously the switching or accepting configuration. By Theorem 3.2 there holds that

$$szl(K_f) + r \times szr(K_f) > szl(K_l) + r \times szr(K_l).$$

We can see that the function f decreases for the sequence of switching configurations, and therefore the value of $f(K_1)$ is the required upper estimation of the number of switches between M and M_1.

ad (b) C is an accepting computation of M if and only if C is a completed starting computation of M_R, and the subautomaton M_1 during C is not started. Since an error sign is inserted by any call of M_1, and only by a call of M_1, the proposition (b) holds. □

Definition 6.2. Let $c \in B, q \in Q, b \in A \cup \{\$\}$.

$$LEFT(M,q,c,b) = \{v \in A^*; (c,q,vb) \Rightarrow^* (\lambda, q_f, b) \text{ by } M \text{ for some } q_f\}.$$

The trace of the computation by M leading from (c,q,vb) to (λ, q_f, b) is denoted by $LETR(M,q,cvb)$.

Computations with the trace equal to $LETR(M,q,cvb)$ are called left-hand side computations on the string cvb from the state q. We can see that the trace of the left-hand side computation on the string cvb from the state q does not depend on the left-hand side context of c and the right-hand side context of b.

We denote by $S_L(M,q,cvb)$ the tree created by the trace $LETR(M,q,cvb)$.

We write

$$S_L(M,q,c,b) = \{S_L(M,q,cvb); v \in LEFT(M,q,c,b)\}.$$

We set

$$RIGHT(M,q,c,b) = \{y \in A^*; (c,q,yb) \Rightarrow^* (cb,q_1,\lambda) \text{ by } M, \text{ for some } q_1\}.$$

The trace of the computation by M leading from (c,q,yb) to (cb,q_1,λ) is denoted by $RITR(M,q,cyb)$.

Computations with the trace equal to $RITR(M,q,cyb)$ are called right-hand side computations on the string cyb from the state q. We can see that the trace of the right-hand side computations on the string cvb from the state q does not depend on the left-hand side context of c and the right-hand side context of b.

We denote by $S_R(M,q,cyb)$ the tree created by the trace $RITR(M,q,cyb)$.

We take

$$S_R(M,q,c,b) = \{S_R(M,q,cyb); y \in RIGHT(M,q,c,b)\}.$$

Definition 6.3. Let c be a headsymbol of M, $b \in A \cup \{\$\}, q = SV(c)$. We set $LEFT(M,c,b) = LEFT(M,q,c,b)$.

We say instead of left-hand side computation on cvb from the state g only left-hand side computation on cvb. The trace of such a computation is denoted as LETR(M,cvb). We can see that this computation finishes in the state $q_f = SE(c)$.

Let $v \in LEFT(M,c,b)$. We denote as $S_L(M,cvb)$ the tree $S_L(M,q,cvb)$, and we write $S_L(M,c,b) = S_L(M,q,c,b)$.

Remark. For a parser M there certainly holds

$$LEFT(M,\#,\$) = L(M), \text{ and } S_L(M,\#,\$) = S_A(M).$$

Definition 6.4. Let c,b be headsymbols of M and $\{q\} = SV(c), \{q_1\} = SV(b)$.

We set $RIGHT(M,c,b) = RIGHT(M,q,c,b)$, $RITR(M,cyb) = RITR(M,q,cyb)$, $S_R(M,cyb) = S_R(M,q,cyb)$. We take $S_R(M,c,b) = S_R(M,q,c,b)$.

Claim 6.5. Let c be a headsymbol, S be from $S_A(M)$, $S_1 \ll S$, S_1 and S_2 be from $S_L(M,c,b)$. By replacing the subtree S_1 by S_2 in S we get a moving-tree S_3 from $S_A(M)$.

Proof. Let the computation C_1 creating S_1 begins in a configuration (xc,q,vby) and ends in a configuration (x,q_1,by). We can see that any computation C_2 creating S_2 starts from a configuration k_1 of the form (zc,q,ubw), and finishes in the unambiguously determined configuration $k_2 = (z,q_1,bw)$.

We can see that the computation C creating S starts in a configuration of the form $(\#,q0,tcvby)$. We show that we can take as S_3 the moving-tree computed by the computation C_3, starting from the configuration $(\#,q_0,tcuby)$. It is obvious that the sequence of instructions used by the computations C until the first visit

on c is a prefix of the trace of C_3 (because of the determimism of M). That means that the configuration $k_3 = (xc, q, uby)$ is the starting configuration of C_3. Therefore the computation C_3 continues from k_3 with a trace equal to the trace of C_2. The configuration reached by C_3 is (x, q_1, by) which is the above mentioned last configuration of C_1. Consequently, the tails of both computations C and C_3 are equal to each other. Since $S \in S_A(M)$, then $S_3 \in S_A(M)$. □

The next corollary expresses the previous fact with the help of traces instead of moving-trees.

Corollary 6.6. Let c be a headsymbol of M $v, z \in LEFT(M, c, b)$, $LETR(M, \#xcvby\$)$ have the form $tr_1, LETR(M, cvb), tr_2$ for some sequences of instructions tr_1 and tr_2. Then the sequence $tr_1, LETR(M, czb), tr_2$ creates a trace of some accepting computation of M.

Claim 6.7. Let c, b be headsymbols, $S \in S_A(M)$, $S_1 \prec S$, and $S_1, S_2 \in S_R(M, c, b)$. Replacing the subtree S_1 by S_2 in S we get a moving-tree $S_3 \in S_A(M)$.

Proof. Let us suppose $S_1 = S_R(M, cvb), S_2 = S_R(M, czb)$ and $S = S_L(M, \#xcvby\$)$. Let S_3 be the moving-tree created by the starting completed computation imposed on $\#xczby\$$. The prefixes of traces used by computations of S and S_3 until the first visit of c are equal to each other. Because the symbol b is a headsymbol the computations on $xcvb$ and $xczb$ reach the same configuration $(\#xcb, SV(b), y\$)$. Therefore the tails of both computations are equal to each other. Since $S \in S_A(M)$, then $S_3 \in S_A(M)$. □

We can see that a similar corollary can be formulated to the Claim 6.7 as to the previous one.

Claim 6.8. Let c be a headsymbol from M, $S_1 \in S_L(M, c, b)$. Then there is $S \in S_A(M)$ such that $S_1 \ll S$.

Proof. By the definition S_1 has on the root the headsymbol c. Since M is a reduced parser, the instruction I which deletes c by the left-hand side computation on cvb, is used in some accepting computation C by M. Let us suppose that the moving-tree computed by C is denoted as S_3. Since I is of the shape $[c, q, b] \to [q_1, DEL]$ there is $S_2 \in S_L(M, c, b)$ such that $S_2 \ll S_3$. By the Claim 6.5 there is $S \in S_A(M)$ such that $S_1 \ll S$. □

The next claim can be demonstrated in a quite similar way.

Claim 6.9. Let c, b be headsymbols from M, $S_1 \in S_R(M, c, b)$. Then there is $S \in S_A(M)$ such that $S_1 \prec S$.

Remark. We can see that the left-hand side and right-hand side computations depend only on the visited and looked items. The previous statements allow to consider any left-hand side or right-hand side computation of M as a surely correct (invariant) computation.

Corollary 6.10. Let c be a headsymbol, $c \in RB$. We can see that the first visit on c by a computation of M_R is always done by the state $SV(c)$ of M. By this observation the following convention is correct. We denote $LEFT(M_R, c, b) = LEFT(M_R, SV(c), c, b)$, $RIGHT(M_R, c, b) = RIGHT(M_R, SV(c), c, b)$, $S_L(M_R, c, b) = S_L(M_R, SV(c), c, b)$, $S_R(M_R, c, b) = S_R(M_R, SV(c), c, b)$.

Theorem 6.11. Let $y \in A^*$, $y = uavbw$, where a is a headsymbol of M from RB, $b \in A$, $v \in LEFT(M, a, b)$. Then $S_L(M, avb) \ll S_A(M_R, \#y\$)$ and $LETR(M_R, \#y\$)$ has the form $t_1, LETR(M, avb), t_2$ for some sequences of instructions t_1 and t_2.

Proof. Since a is a headsymbol of M from RB, then a left-hand side computation by M is performed on avb. This observation implies the prooved assertion. □

Theorem 6.12. Let $y \in A^*$, $y = uavbw$, where a is a headsymbol of M, $a \in RB$ and $v \in RIGHT(M, a, b)$. Then $S_R(M, avb) \prec S_A(M_R, \#y\$)$, and $RITR(M_R, \#y\$)$ has the form $t_1, RITR(M, avb), t_2$ for some t_1 and t_2.

Proof. Since $a \in RB$, $v \in RIGHT(M, a, b)$ we can see that the first visit on a is done by M, therefore the whole computation on avb is a right-hand side computation of M. This implies the statement. □

Theorem 6.13. Let c be an inserted headsymbol by Mor a headsymbol from RB, $S_1 \in S_A(M_R)$ and $S_2 \ll S_1$, where $S_2 \in S_L(M_R, c, b)$ does not contain any error sign. Then $S_2 \in S_L(M, c, b)$.

Proof. Since c is a headsymbol of M from RB or an inserted symbol, and S_2 does not contain any error sign, then all nodes of S_2 are visited and therefore created by M. Thus $S_2 \in S_L(M, c, b)$. □

Remark. A similar statement to the previous one can be formulated for a string-tree from $S_R(M_R, c, b)$.

The previous statements give us guarantees about correctness of the left-hand side and right-hand side computations. Left and right-hand side computations are continuous computations considering time and topology as well. In the sequel we outline some possibilities to give similar guarantees for some computations discontinuous in time. We will take the advantage of the topological continuity of these computations. In the next definition we introduce tools to handle such computations.

Definition 6.14. Let $S2_i$ for $i \in \{1, \ldots, n\}$, and S_1 be string-trees, where $S2_i \ll S_1$ for $i \in \{1, \ldots, n\}$, and any two different string-trees from $\{S2_1, \ldots, S2_n\}$ do not have any common node, and $S_{L_1} = S_1 - S2_1, S_{L_2} = S_{L_1} - S2_2, \ldots, S_{L_n} = S_{L_{n-1}} - S2_n$. Then we can write $S_{L_n} = S_1 - \{S2_1, \ldots, S2_n\}$.

Let $S_L(M, d, e)$ be nonempty, $SV(d) = q$, $SE(d) = q_1$. We denote by $S_{L_M}(M_R, d, e)$ the subset of $S_L(M_R, d, e)$ such that any computation creating some member of $S_{L_M}(M_R, d, e)$ starts from the state q and finishes in the state q_1.

Theorem 6.15. Let $S_1 \in S_R(M_R, c, b)$ and $S2_i \in S_{L_M}(M_R, d_i, e_i)$ for i between 1 and n, and the tree $S_3 = S1 - \{S2_1, \ldots, S2_n\}$ does not contain any error or special symbol. Then S_3 is a subtree of some moving-tree from $S_R(M, c, b)$.

Proof. Since $S_R(M_R, c, d)$ is defined and nonempty, c is a headsymbol from RB. Let C_1 be the computation creating S_1 and $C2_i$ be the computations creating $S2_i$. $C2_i$ are subcomputations of C_1. C_1 starts from the state $SV(c)$ of M. Therefore any computation of M_1 can be awakened only inside of some computation $C2_i$. By our assumptions there are left-hand side computations of M, which can be placed instead of computations $C2_i$ into C_1. By this operations we can get a new computation $C3$. Since S_3 does not contain any error or special symbols, all edges from S_3 are created by M. Thus C_3 is a right-hand side computation by M. □

Remark. We can see that we can get some other variants to the previous statement.

An important thing is to see that the number of nested (or iterated) separated subtrees computed by M inside a tree from $S_A(M_R)$ is in general not bounded by any constant.

We can also see that the size of any tree from $S_A(M_R)$ is linearly bounded comparing to the length of the input, which means that the number of error symbols is also linearly bounded.

Remarks about headsymbols. An input headsymbol a, for which there is a symbol b such that $LEFT(M, a, b)$ is an infinite language, is more suitable for our construction than some headsymbol c, for which the sets $\{v; v \in LEFT(M, c, b)$ for some $b\}$ are empty.

It is easy to check whether a symbol is a headsymbol of M. By the definition it is sufficient to check the set of instructions of M.

7. EXAMPLES

Example 7.1. For an automaton $MA = (Q_A, A, A \cup \{\#, \$\}, IT_A, q_0)$, which represents the reduced equivalent of the automaton M_A from Example 2.2, we construct by the construction 5 the automaton $MA_R = (QA_R, A, A \cup \{\#, \$, @, \&\}, IT_{AR}, q_0)$. We choose for this construction $LB = \{\#, +, =\}$, $RB = \{\$\}$, $DLL = \{(\#, \$), (=, \$), (+, \$)\}$, and MRB equal to the empty set. The symbol @ is the single error sign, the

symbol & is the single special sign. Despite the empty set MRB the MA_R seems to have a reasonable abbility of the recovery.

IT_A is described by the following schemes:

(1a) $(\{\#, +, =\}, q_0, \{a, b, \}) \rightarrow (q_0, MVR)$
(1b) $(\{, a, b, \}, q_0, \{+, =\}) \rightarrow (q_0, MVR)$
(2) $(\{a, b\}, q_0, \{\$\}) \rightarrow (q_1, DEL)$
(3) $(\{+\}, q_1, \{\$\}) \rightarrow (q_a, DEL)$
(4) $(\{a, b\}, q_a, \{\$\}) \rightarrow (q_1, DEL)$
(5) $(\{=\}, q_1, \{\$\}) \rightarrow (q_=, DEL)$
(6) $(\{a, b\}, q_=, \{\$\}) \rightarrow (q_\#, DEL)$
(7) $(\{\#\}, q_\#, \{\$\}) \rightarrow (q_0, DEL)$;

– the remainder of IT_{AR} :

(9)	$(\{\#, +, =\}, q_0, \{+, =, \$\}) \rightarrow (kvi, IN(@))$	command 1
(10)	$(\{\#\}, q_1, \{\$\}) \rightarrow (kvi, IN(@))$	command 1
(11)	$(\{a, b\}, q_0, \{a, b\}) \rightarrow (kvi, IN(@))$	command 1
(12)	$(\{=, +, a, b\}, q_\#, \{\$\}) \rightarrow (kvi, IN(@))$	command 1
(13)	$(\{@, a, b, +, =\}, kvi, \{a, b, +, =\}) \rightarrow (kvi, MVR)$	command 2
(13a)	$(\{a, b, +, =\}, kvi, \{\$\}) \rightarrow (kvi, DEL)$	command 3
(14)	$(\{@\}, kvi, \$) \rightarrow (kvd, DEL)$	command 3
(15a)	$(\{+\}, kvd, \{\$\}) \rightarrow (kd, IN(\&))$	command 7
(15b)	$(\{\&\}, ka, \{\$\}) \rightarrow (ka, DEL)$	command 7
(15c)	$(\{+\}, ka, \{\$\}) \rightarrow (q_a, DEL)$	command 7
(16a)	$(\{=\}, kvd, \{\$\}) \rightarrow (k =, IN(\&))$	command 7
(16b)	$(\{\&\}, k =, \{\$\}) \rightarrow (k =, DEL)$	command 7
(16c)	$(\{=\}, k =, \{\$\}) \rightarrow (q_=, DEL)$	command 7
(17)	$(\{a, b\}, kvd, \{\$\}) \rightarrow (kvd, DEL)$	command 8
(18a)	$(\{\#\}, kvd, \{\$\}) \rightarrow (k\#, IN(\&))$	command 7
(18b)	$(\{\&\}, k\#, \{\$\}) \rightarrow (k\#, DEL)$	command 7
(18c)	$(\{\#\}, k\#, \{\$\}) \rightarrow (q_0, DEL)$	command 7

The automaton MA_R computes to the input string with sentinels $\#a = aa + \$$ the following moving-tree (in the /-description) computes to the input string with the sentinels $\#a = aa + \$$:

$$\#_0/a_1/ =_2 /(a_3/@_4/a_5/+_6, \&_7).$$

We can see that the previous tree has only one correct component, namely $\#_0/a_1/ =_2$, and one incorrect component. We can see that the incorrect part begins after the node $=_2$, since the special node $\&_7$ directly depends on it.

Let us take

$$\#_0/a_1/ =_2 /@_3/(a_4/ =_5 /(a_6/@_7/a_8, \&_9), \&_{10}).$$

This tree computed by MA_R has also only one correct component, but two separate incorrect components corresponding to two different errors.

Example 7.2. We construct a robust parser $MC' = (QC', A, A \cup \{\#, \$, C, @, \&\}, ITC', q_0)$ to the automaton M_C from the Example 2.2. We choose for this construction $LB = \{\#, C\}$, $RB = \{+, \$\}$, $DLL = \{(\#, \$), (C, \$)\}$, $MRB = \{(C, +)\}$. The symbol @ is the single error sign, the symbol & ió the single special sign.

IT_C is described by the sets of schemes:

(1)	$(\{\#\}, q_0, \{a, b\}) \to (q_r, MVR)$
(2)	$(\{a, b\}, q_r, \{=\}) \to (q_=, MVR)$
(3)	$(\{=\}, q_=, \{a, b\}) \to (q, DEL)$
(4)	$(\{a, b\}, q, \{a, b\}) \to (q_=, IN(C))$
(5)	$(\{C\}, q_=, \{a, b\}) \to (q_a, MVR)$
(6)	$(\{a, b\}, q_a, \{+\}) \to (q_+, DEL)$
(7)	$(\{a, b\}, q_a, \{\$\}) \to (q_a, DEL)$
(8)	$(\{C\}, q_+, \{+\}) \to (q_+, MVR)$
(9)	$(\{+\}, q_+, \{a, b\}) \to (q_2, DEL)$
(10)	$(\{C\}, q_2, \{a, b\}) \to (q_a, MVR)$
(11)	$(\{C\}, q_a, \{\$\}) \to (q_1, DEL)$
(12)	$(\{a, b\}, q_1, \{\$\}) \to (q_\$, DEL)$
(13)	$(\{\#\}, q_\$, \{\$\}) \to (q_0, DEL)$

The rest of instructions of ITC':

(14)	$(\{\#\}, q_0, \{=, +, \$\}) \to (kvi, IN(@))$	command 1
(15)	$(\{a, b\}, q_r, \{+, a, b, \$\}) \to (kvi, IN(@))$	command 1
(16)	$(\{=\}, q_=, \{=, +, \$\}) \to (kvi, IN(@))$	command 1
(17)	$(\{+\}, q_+, \{+, =, \$\}) \to (kvi, IN(@))$	command 1
(18)	$(\{a, b\}, q_a, \{a, b, =\}) \to (kvi, IN(@))$	command 1
(19)	$(\{@, a, b, C, =\}, kvi, \{a, b, =\}) \to (kvi, MVR)$	command 2
(20)	$(\{a, b, =\}, kvi, \{\$, +\}) \to (kvd, DEL)$	command 3
(21)	$(\{@, \&\}, kvd, \{\$, +\}) \to (kg, DEL)$	command 3
(22)	$(\{@\}, kvi, \{\$, +\}) \to (kg, DEL)$	command 3
(23)	$(\{C, a, b, =\}, kvd, \{\$, +\}) \to (kvd, DEL)$	command 3
(24)	$(\{a, b\}, kg, \{\$, +\}) \to (kg, DEL)$	command 3
(25a)	$(\{C\}, kg, \{+\}) \to (k+, IN(\&))$	command 4
(25b)	$(\{\&\}, k+, \{+\}) \to (k+, DEL)$	command 4
(25c)	$(\{C\}, k+, \{+\}) \to (q_+, MVR)$	command 4
(26a)	$(\{C\}, kg, \{\$\}) \to (k1, IN(\&))$	command 7
(26b)	$(\{\&\}, k1, \{\$\}) \to (k1, DEL)$	command 7
(26c)	$(\{C\}, k1, \{\$\}) \to (q_1, DEL)$	command 7
(27)	$(\{\#\}, \{kvd, q_1\}, \{\$\}) \to (kin, IN(\&))$	command 7
(28)	$(\{\#\}, kg, \{+, \$\}) \to (kin, IN(\&))$	command 7,4
(28a)	$(\{\&\}, kin, \{\$\}) \to (kin, DEL)$	command 7
(28b)	$(\{\#\}, kin, \{\$\}) \to (q_0, DEL)$	command 7
(29)	$(\{\&\}, kin, \{+\}) \to (q_+, MVR)$	command 5
(30)	$(\{\&\}, q_2, \{\$\}) \to (kg, DEL)$	command 3
(31)	$(\{\&\}, q_2, \{a, b\}) \to (kvi, MVR)$	command 2

The automaton MC' to the input string with sentinels $\#a = aa + \$$ computes the following moving-tree:

$$\#_0/a_1/(=_2, C_3/(a_4/@_5/a_6, \&_7, +_8/@_9, \&_{10}).$$

In this tree two errors are exactly localized and one correct component is separated.

If we add + to LB, and $(+, \$)$ to DLL then we get an enhanced robust parser which separates the correct component enhanced by $+_{\ldots}$ comparing to the previous moving-tree.

8. CONCLUDING REMARKS

Our effort was to describe an algorithmizable construction of a robust parser to a deterministic reduced one. We give the possibility to choose between construction of a big parser which is more sensitive to separating correct and incorrect components, or a smaller parser with a low sensitivity. The sensitivity and the size of the resulting robust parser depends on our choice of the sets LB, RB, DLL and MRB.

We can see that the size of the robust parser is usually much greater than the size of the original reduced parser. That is the reason why the algoritmization of such a construction can be a real practical help.

This paper has arisen as a modification of the technical report [6]. The idea to use list automata to model parsers has come out from [8], where list automata represent parsers suitable for natural languages. A similar notion as the moving-tree was introduced in [7]. Some basic information about the error recovery techniques can be found in [2] and [9].

The error recovery method from [1] can be formulated also with the help of list automata. That makes it possible to extend the method from [1] by our ideas in such a way that we get a fully robust parser. In this way we can get broader guarantees than we have offered in this paper.

ACKNOWLEDGEMENT

I am very thankful to Zuzana Dobeš and to Jaromír Matas for their recommendations.

The work on this paper is a part of the project "Automata for Separation of Syntactically Correct Structures from Syntactically Incorrect Structures" supported by the Grant Agency of the Czech Republic under Grant No. 201/96/0195.

(Received May 23, 1996.)

REFERENCES

[1] M. Chytil and J. Demner: Panic Mode without Panic. In: Automata, Languages and Programming (Lecture Notes in Computer Science 267). Springer-Verlag, Berlin – Heidelberg – New York 1987.

[2] J. Lewi et al: The ELL(1) Parser Generator and the Error Recovery Mechanism. Acta Inform. *10* (1978), 209–228.

[3] J. Matas: Construction of Reduced and l–Corrected List Automata (in Czech). Diploma Thesis, Department of Theoretical Computer Science, Faculty of Mathematics and Physics, Charles University, Prague 1993.

[4] Nguyen Xuan Dung: On Decidability of Skeletal Sets (in Czech). Ph. D. Thesis, Department of Theoretical Computer Science, Faculty of Mathematics and Physics, Charles University, Prague 1988.

[5] M. Plátek: Independent error messages. In: SOFSEM'91, Czechoslovak Society for Computer Science, Jasná pod Chopkom 1991, pp. 257–260.
[6] M. Plátek: Syntactic Error Recovery with Formal Guarantees I. Technical Report No. 100, Department of Theoretical Computer Science, Faculty of Mathematics and Physics, Charles University, Prague 1992.
[7] M. Plátek and J. Vogel: Deterministic list automata and erasing graphs. The Prague Bulletin of Mathematical Linguistics *45* (1986), 27–50.
[8] M. Plátek and B. Tichá: List automata as dependency parsers. The Prague Bulletin of Mathematical Linguistics *50* (1988), 71–88.
[9] N. Wirth: Algorithms + Data Structures = Programs. Prentice–Hall, Englewood Cliffs, NJ 1975.

Martin Plátek, CSc., Katedra teoretické informatiky, Matematicko-fyzikální fakulta Univerzity Karlovy (Department of Theoretical Computer Science, Faculty of Mathematics and Physics – Charles University), Malostranské náměstí 25, 118 00 Praha 1. Czech Republic.