SIMPLE ERROR RECOVERY SCHEME
FOR OPTIMIZED LR-PARSERS

Jerzy S. Krol

TR 81-456
March 1981

Department of Computer Science
Cornell University
Ithaca, New York 15843

Simple Error Recovery Scheme for Optimized LR-Parsers

Jerzy S. Krol

Cornell University

ABSTRACT

A new, simple and effective method for syntactic error recovery in optimized (reduced) LR-parsers is presented. This method, called the Simple Recovery and Correction scheme, is phrase oriented and performs local and some form of global context correction. The error handling mechanism is driven by information obtainable from an LR-parser decision table.

The formal basis for the method is the concept of synchronizing triple. A theoretical characterization of synchronizing triples is given and algorithms for direct extraction of recovery control information are presented.

As a part of the SRC scheme a simplified method for the organization of LR-parser forward moves is introduced.

In the last part of the paper the performance of the SRC scheme is illustrated in a specific case and implementation problems are discussed.

Key Words and Phrases: LR-parsing, syntactic error recovery

CR Categories: 4.2, 4.12, 5.23

Author's address: Department of Computer Science, Cornell University, 405 Upson Hall, Ithaca, N.Y. 14853.

1. Introduction

One particularly desirable feature of a compiler is its ability to detect all program errors which can be found in a single run. Simultaneously each formal parsing algorithm can only analyze strings from the language, or discover that a given string does not belong to this language. There exists no simple and natural generalization of formal parsing methods for all strings of symbols from the language vocabulary. The reason for this is that error detection is equivalent to corroboration that a given string does not belong to the language; detecting subsequent errors in this string involves testing whether the string belongs to the language and not always expecting negative answers. As a consequence, to detect any further errors we have to use some heuristic methods called error recovery and/or correction schemes.

The purpose of a syntactic error recovery scheme is to allow the parsing of an imperfect program to continue past a point of error, so as to discover as many additional errors as possible.

In contrast, an error correction scheme attempts to transform the erroreous part of an input string into a similar and syntactically correct one. In nearly all such schemes, it is implicitly assumed that the error is local in its extent and may be corrected by inserting, deleting or replacing a single syntactic token.

Research, to date, on methods for handling programs with errors is reflected in more then fifty published papers (see the references and the bibliography [3]).

Actually, for LR-parsers [1,4,10], there exist error recovery methods with high quality error corrections and great time or space complexity [2,5,7,9,13,16-19], as well as very simple "panic mode" methods [1,8,14] which suffer from discarding large portions of a source program and generating spurious errors. There are no methods with intermediate features (moderate complexity, satisfactory quality of error recovery, simple implementation) that also give the possibility of fully automatic generation of recovery procedures.

In this paper we present a simple, table driven error recovery method for optimized LR-parsers[1], which involves both local and certain global context

1) Note that LR-parser optimizations, such as removing LR(0)-reduce states, have implications for error recovery, since the optimized parser does no checking for illegal lookahead symbols. This could cause one or more reductions to be made before the illegal symbol is detected.

corrections.

The main goal of our method, which we call Simple Recovery and Correction scheme -- in short, SRC -- is not to repair the whole program, but to continue parsing and produce a maximum number of useful diagnoses with a minimum of noise. This method is an extension of the concept of synchronizing triples, to optimized LR-parsers [1,10], which was introduced in [11] for precedence parsers.

As part of the SRC scheme, we describe a simple method for restarting an LR-parser after an error detection point, enabling the forward move, which attempts to parse, ahead to discover some right context upon which the error correction can be done.

2. Notation and definitions

We assumed that the reader is familiar with the basic concepts of LR parsing [1].

We use the notation from [1,4,10]. In particular, $G=(N,\Sigma,P,S)$ is a CFG where N and Σ are finite, nonempty sets of nonterminal and terminal symbols; $P \subset N \times (N \times \Sigma)^*$ is a set of productions, (we write $A \rightarrow \alpha$ instead of (A, α)), and $S \in N$ is a distinguished symbol, called the sentence (or start) symbol.

We also use the following convention:

S, A, B, C,...	- denote elements of N;
a, b, c,...	- denote elements of $\Sigma \cup \{\perp\}$;
U, V,..., Z	- denote elements of $N \cup \Sigma \cup \{\perp\}$;
u, v,..., z	- denote elements of $(\Sigma \cup \{\perp\})^*$;
$\alpha, \beta, \gamma,...$	- denote elements of $(N \cup \Sigma \cup \{\perp\})^*$;
e	- denotes an empty string of grammar symbols;
$\perp$	- denotes the endmarker , $\perp \notin (N \cup \Sigma)$.

All the grammars considered in the following are assumed to be context -free and reduced, i.e. cycle-free and with no useless symbols.

Following [10] we use a specific formalization of an LR-parser, i.e. any one symbol look-ahead LR-parser, such as an LR(1), LALR(1) or SLR(1). This formalization resembles one introduced in [4].

As has been mentioned we are interested in the construction of a simple and practical error recovery and correction method which is applicable to optimized LR-parsers (with removed LR(0)-reduce states [1,10] -- in short, reduced LR-parsers).

Definition 2.1. A reduced LR-parser for CFG $G=(N,\Sigma,P,S)$ is a modified LR(1)-automaton $LRA'(G)=(Q,N,\Sigma,P,q_0,\text{next},\text{reduce},\text{read})$ such that

Q - is a finite, nonempty set of states;
N,Σ,P - are as in the grammar G;
$q_0\in Q$ - is a distinguished start state;
$\text{next}: Q\times(N\cup\Sigma\cup\{\perp\})\to(Q\cup\{\text{stop}\})$ - is a partially defined state transition function;
$\text{reduce}: Q\times(N\cup\Sigma\cup\{\perp\}\to 2^P$ - is a reduce function;
$\text{read}: Q\times(N\cup\Sigma\cup\{\perp\})\to\{\text{true},\text{false}\}$ - is a predicate which implies that the next terminal input symbol must be shifted during a phrase reduction, i.e. $\text{read}(p,X)=\text{false}\ \forall X\in N$.

The complete definition of a reduced LR-parser and a method for its direct construction for a given grammar has been presented in [10][2]. In this paper we give only the operational semantics of a deterministic reduced LR-parser in terms of its moves.

Definition 2.2. A reduced LR-parser is deterministic if for all $q\in Q$, X we have

$$\text{card}(\text{reduce}(q,X))\leq 1 \wedge ((q,X)\in\text{dom next} \Rightarrow \text{reduce}(q,X)=\emptyset).$$

A configuration of an LR-parser is a pair $([\alpha],\beta)$ in $Q^+\times(N\cup\Sigma\cup\{\perp\})^+$. Its first part is the parse stack, whose top is on the right, and the second is the input tape.

To simplify the description of LR-parser moves we also introduce two additional distinquished configurations: **stop** and **error.**

A move by a deterministic reduced LR-parser is represented by a binary

2) This method utilizes the SLR(1) technique but may be easily extended to the LALR(1) technique [1,4].

relation $\vdash$ on the set of parser configurations.

Definition 2.3. A relation $\vdash$ on configurations of a deterministic reduced LR-parser is defined as follows:

$\forall w : (q_0 \cdots q_n , Xw)$

1.	$\vdash (q_0 \cdots q_n q_{n+1} , w)$	if $reduce(q_n, X) = q_{n+1}$;
2.	$\vdash (q_0 \cdots q_{n-k} , Aw)$	if $reduce(q_n, X) = \{A \to \alpha\} \wedge (read(q_n, X) \vee X \in N)$, $k = \lvert\alpha\rvert - 1$;
3.	$\vdash (q_0 \cdots q_{n-k} , AXw)$	if $reduce(q_n, X) = \{A \to \alpha\} \wedge \neg(read(q_n, X) \vee X \in N)$, $k = \lvert\alpha\rvert$;
4.	$\vdash$ **stop**	if $next(q_n, X) = stop$;
5.	$\vdash$ **error**	if $reduce(q_n, X) = \emptyset \wedge (q_n, X) \notin dom\, next$.

During the analysis of an input string the parser proceeds from one configuration to another by processing the next input symbol (a shift move - (1)) or by reducing a phrase with or without reading the next input symbol (a reduce move - (2) and (3)). The parser begins its moves in the initial configuration $(q_0, w\bot)$ and finishes in the accepting configuration $(q_0 q, \bot)$, where $q = next(q_0, S)$ and $next(q, \bot) = stop$, or in an error configuration $(q_0 \cdots q_n, Xw)$ such that $reduce(q_n, X) = \emptyset \wedge (q_n, X) \notin dom\, next$.

Observe that, for technical reasons, we have extended the shift move to permit shifts on nonterminal symbols as well.

For convenience we extend function next to strings from $(N \cup \Sigma \cup \{\bot\})^*$ and introduce a predicate R.

Let $\alpha = X_1 \cdots X_n$, $n \geq 0$. Then $next(p, \alpha) = q$ if $\exists p_0, \ldots, p_n \in Q : (p_0 = p \wedge p_n = q \wedge (\forall i : 0 < i \leq n : p_i = next(p_{i-1}, X_i)))$.

An alternative notation for the above sequence $p_0 \cdots p_n$ of states is $[p : \alpha]$, or simply $[\alpha]$ if $p = q_0$.

Let R be a predicate on $Q \times (N \cup \Sigma \cup \{\bot\})^*$ such that

$$R(p,X) = ((p,X) \in dom\, next \vee reduce(p,X) \neq \emptyset \wedge (read(p,X) \vee X \in N)).$$

Lemma 2.4. $R(p,X) \Leftrightarrow (\forall \alpha : next(q_0, \alpha) = p\ \exists \beta : (\forall w : ([\alpha], Xw) \vdash^+ ([\beta], w)))$.

Proof. The lemma follows directly from definition 2.3 and the construction of reduced LR-parser.$\square$

3. Detecting and diagnosing syntactic errors

In a reduced LR-parser[3)] detecting and diagnosing errors may be implemented fairly easily. Let $ERR : Q \times (N \cup \Sigma \cup \{\bot\}) \rightarrow \{true, false\}$ be a predicate such that

$$ERR(q,X) = ((q,X) \notin dom\, next \wedge reduce(q,X) = \emptyset).$$

From definition 2.3 and the methods for LR-parser construction [1,4,10], it is evident that the reduced LR-parser detects a syntactic error in a state q for a terminal symbol a if ERR(q,a) is true, and that during the parser activities only such error pairs (q,a) can be tested for which one of the following four conditions is true:

(i) $q = q_0$;

(ii) $\exists (p \in Q\, ,\ b \in \Sigma) : q = next(p,b)$;

(iii) $\exists (p,r \in Q\, ,\ (A \rightarrow \alpha X) \in P) : q = next(p,A) \wedge r = next(p,\alpha) \wedge (A \rightarrow \alpha X) \in reduce(r,X) \wedge (read(r,X) \vee X \in N)$;

(iv) $\exists (p,r \in Q\, ,\ (A \rightarrow \alpha) \in P) : q = next(p,A) \wedge r = next(p,\alpha) \wedge (A \rightarrow \alpha) \in reduce(r,a) \wedge \neg read(r,a)$.

Conditions (i) - (iii) mean that state q is reachable regardless of the symbol a; condition (iv) implies that state q is reachable in the course of phrase reduction determined by symbol a.

The determination of such a set of proper error pairs makes it possible to introduce a clear error diagnosis scheme. It can contain some preconstructed set of error messages, or these messages can be generated on-line. In the latter case the message should contain an error token (a) as well as a set $\{a : \neg ERR(q,a)\}$ of terminal symbols expected in state q. In practice, the realization of the above postulate sometimes leads to confusing messages. The reason is that there are usually many symbols expected in a lookahead state[4)] and thay may be misleading. Hence it is more convenient [5] to differentiate forms of messages by listing a commonly not too large set of expected symbols only in a read state and giving the information that a symbol "a" is unexpected if (q,a) is an error pair and q is a lookahead state.

---

3) In the following the term "reduced LR-parser" will denote a deterministic reduced LR-parser.

4) A state $q \in Q$ is a lookahead state if $\exists a : reduce(q,a) \neq \emptyset \wedge \neg read(q,a)$ and is a read state otherwise.

After an error detection the parser must recover and continue to parse in order to detect further errors. Consequently it should give not only an error message describing precisely the nature and the location of the error, but it should explain the recovery action. Otherwise the user is not able to distinquish between his errors and those caused by inadeqate recovery from previous ones [18]. We will discuss this problem later.

4. Synchronizing triples

The basis for our Simple Recovery and Correction scheme is the concept of synchronizing triples.

To introduce synchronizing triples we informally discuss a special case.

Let $(p,Z,a) \in Q \times (N \cup \Sigma) \times (\Sigma \cup \{\perp\})$ be a (synchronizing) triple such that

$$(\forall \alpha : \text{next}(q_0, \alpha) = p : (\exists u : S\perp \Rightarrow^{*}_{R} \alpha Zau))$$

and (p,a) is an error pair for the given LR-parser. If the parser has detected the error in a configuration $([\alpha], aw)$, where $\text{next}(q_0, \alpha) = p$, then it is possible to correct it by inserting the symbol Z before a, so that the latter symbol can be properly processed (shifted).

It is also possible (and sometimes useful) to weaken the concept of synchronizing triples so that, in the above case, Z will always be shifted and the symbol a will be shifted for at least one prefix α of a right sentential form.

Definition 4.1. A triple $(p,Z,X) \in Q \times (N \cup \Sigma \cup \{\perp\})^2$ is called
a. a synchronizing triple if

$$\forall \alpha : \text{next}(q_0, \alpha) = p : (\exists \beta : (\forall w : ([\alpha], ZXw) \vdash^{+} ([\beta], w)));$$

b. a weak synchronizing triple if both of the following conditions are fulfiled:

i) $\forall \alpha : \text{next}(q_0, \alpha) = p : (\exists \beta : (\forall w : ([\alpha], ZXw) \vdash^{+} ([\beta], Xw)));$
ii) $\exists \alpha : \text{next}(q_0, \alpha) = p,\ \exists \beta : (\forall w : ([\alpha], ZXw) \vdash^{+} ([\beta], w)).$

If (p,Z,X) is a (weak) synchronizing triple then we call:

(p,X) - a synchronizing pair,
X - a synchronizing symbol,
Z - a (weak) correcting symbol for the pair (p,X).

In the following, ST (WST) will denote the set of all (weak) synchronizing triples for a given reduced LR-parser; from definition 4.1 we have $ST \subset WST$.

The above definition gives us a precise characterization of synchronizing triples, but it is not clear how to apply it effectively to determine sets ST and WST of such triples, given a grammar G. Consequently we need some additional notions which lead to an algorithm for direct construction of those sets.

Definition 4.2. A generalized state transition function with shift is a mapping

$$NEXT' : Q \times (N \cup \Sigma \cup \{\perp\}) \rightarrow 2^{Q \cup \{stop\}}$$

such that

$$\begin{aligned} NEXT'(p,X) = \{q : \exists (k \geq 0; p_0,\ldots,p_k \in Q \cup \{stop\};\ X_0,\ldots,X_k;\ \alpha_1,\ldots,\alpha_k) : \\ q = next(p_k, X_k) \wedge (\forall i : 0 \leq i < k : p_i = next(p_{i+1}, \alpha_{i+1}) \wedge \\ reduce(p_i, X_i) = \{X_{i+1} \rightarrow \alpha_{i+1} X_i\} \wedge \\ (read(p_i, X_i) \vee X_i \in N)) \wedge p_0 = p \wedge X_0 = X\}. \end{aligned}$$

Lemma 4.3. $NEXT'(p,X) = \{q : (\forall \alpha : next(q_0, \alpha) = p)\ (\exists \beta : next(q_0, \beta) = q) :$
$(\forall w : ([\alpha], Xw) \vdash^{+} ([\beta], w))$
through transitions of the type (1) or (2) from definition 2.3$\}$.

Proof. This lemma follows directly from definitions 2.3 and 4.2 and the construction of the reduced LR-parser.□

Corollary. $NEXT'(p,X) \neq \emptyset \Leftrightarrow R(p,X)$.

Definition 4.4. A generalized state transition function without shift is a mapping

$$NEXT'' : Q \times (N \cup \Sigma \cup \{\perp\}) \rightarrow 2^{Q}$$

such that

$$\begin{aligned} NEXT''(p,X) = \{q : \exists (k > 0;\ p_0,\ldots,p_k,\ p'_1,\ldots,p'_k \in Q;\ A_1,\ldots,A_k;\ \alpha_1,\ldots,\alpha_k) : \\ (\forall i : 0 \leq i < k : p_i = next(p'_{i+1}, \alpha_{i+1}) \wedge \\ p_{i+1} \in NEXT'(p'_{i+1}, A_{i+1}) \wedge reduce(p_i, X) = \{A_{i+1} \rightarrow \alpha_{i+1}\} \wedge \\ \neg(read(p_i, X) \vee X \in N)) \wedge p = p_0 \wedge q = p_k \wedge \end{aligned}$$

$(reduce(q,X) = \emptyset \vee read(q,X) \vee X \in N)\}$.

Lemma 4.5. $NEXT''(p,X) = \{q : (\forall \alpha : next(q_0, \alpha) = p)\ (\exists \beta : next(q_0, \beta) = q) :$
$reduce(p,X) \neq \emptyset \wedge \neg(read(p,X) \vee X \in N) \wedge$
$(\forall w : ([\alpha],Xw) \vdash^{+} ([\beta],Xw)) \wedge$
$(reduce(q,X) = \emptyset \vee read(q,X) \vee X \in N)\}$.

Proof. Similar to the proof of lemma 4.3.□

Corollary. $NEXT''(p,X) = \emptyset\ \forall X \in N$.

We are now ready to present the algorithms for direct construction of generalized state transition functions. To simplify their description, we define two auxiliary relations μ and $\mathcal{V}$ on a set $Q \times (N \cup \Sigma \cup \{\perp\})$.

Definition 4.6.

(i) $(p,X)\mu(q,Y) \Leftrightarrow \exists \alpha : next(q,\alpha) = p \wedge$
$reduce(p,X) = \{Y \to \alpha X\} \wedge (read(p,X) \vee X \in N)$;

(ii) $(p,X)\mathcal{V}(q,Y) \Leftrightarrow \exists \alpha : next(q,\alpha) = p \wedge$
$reduce(p,X) = \{Y \to \alpha\} \wedge \neg(read(p,X) \vee X \in N)$.

The use of these relations and the following notion gives us the possibility for a simple and compact redefinition of functions NEXT' and NEXT''.

Let Q and R be binary relations on a set S. The composition of Q and R, denoted by QR, is $\{(x,z) : (\exists y \in S) : xRy \wedge yQz\}$. The completion of R, denoted by comp(R), is $\{(x,y) : xR^{*}y \wedge (\neg\exists z \in S) : yRz\}$.

Lemma 4.7.

(i) $q \in NEXT'(p,X) \Leftrightarrow (p,X)\ next\ comp(\mu)\ q$;

(ii) $q \in NEXT''(p,X) \Leftrightarrow (p,X)\ comp(next \times \{X\})\ comp(\mu)\ \mathcal{V}\ (q,X)$;

where next may be interpreted as a relation on $Q \times (N \cup \Sigma \cup \{\perp\}) \times Q$, equivalent to a partial function next.

Proof. The lemma follows directly from definitions 4.2, 4.4 and 4.6.□

For practical computations of values of generalized state transition functions we can also use the following algorithms.

Algorithm 1.

Input: $p \in Q$, X.
Output: A set NEXT = NEXT'(p,X).
Method:

$$
\begin{array}{l}
\textbf{if } (p,X) \in \mathrm{dom\,next} \rightarrow \mathrm{NEXT} := \{\mathrm{next}(p,X)\} ; \\
\ \square\ (p,X) \notin \mathrm{dom\,next} \rightarrow \\
\qquad \mathrm{NEXT} := \emptyset ;\ \mathrm{UNTESTED} := \{(p,X)\} ;\ \mathrm{TESTED} := \emptyset ; \\
\qquad \textbf{do } \mathrm{UNTESTED} \neq \emptyset \rightarrow \\
\qquad\quad \mathrm{PAIRS} := \bigcup_{(q,Z) \in \mathrm{UNTESTED}} \{(s,A) : (q,Z)\, \mu\, (s,A)\} ; \\
\qquad\quad \mathrm{NEXT} := \mathrm{NEXT} \cup \{t : \exists (s,A) \in \mathrm{PAIRS} : t = \mathrm{next}(s,A)\} ; \\
\qquad\quad \mathrm{TESTED} := \mathrm{TESTED} \cup \mathrm{UNTESTED} ; \\
\qquad\quad \mathrm{UNTESTED} := \{(s,A) \in \mathrm{PAIRS} : (s,A) \notin \mathrm{dom\,next}\} \setminus \mathrm{TESTED} \\
\qquad \textbf{od} \\
\ \textbf{fi}
\end{array}
$$

Algorithm 2.

Input: $p \in Q$, X.
Output: A set NEXT = NEXT''(p,X).
Method:

$$
\begin{array}{l}
\mathrm{NEXT} := \emptyset ;\ \mathrm{UNTESTED} := \{p\} ;\ \mathrm{TESTED} := \emptyset ; \\
\textbf{do } \mathrm{UNTESTED} \neq \emptyset \rightarrow \\
\quad \mathrm{PAIRS} := \bigcup_{q \in \mathrm{UNTESTED}} \{(s,A) : (q,X)\, \nu\, (s,A)\} ; \\
\quad \mathrm{NEXT} := \mathrm{NEXT} \cup \{t : \exists (s,A) \in \mathrm{PAIRS} : t \in \mathrm{NEXT}'(s,A) \wedge \\
\qquad (\mathrm{reduce}(t,X) = \emptyset \vee \mathrm{read}(t,X) \vee X \in N)\} ; \\
\quad \mathrm{TESTED} := \mathrm{TESTED} \cup \mathrm{UNTESTED} ; \\
\quad \mathrm{UNTESTED} := \{t : \exists (s,A) \in \mathrm{PAIRS} : t \in \mathrm{NEXT}'(s,A) \wedge \\
\qquad \mathrm{reduce}(t,X) \neq \emptyset \wedge \neg(\mathrm{read}(t,X) \vee X \in N)\} \setminus \mathrm{TESTED} \\
\textbf{od}
\end{array}
$$

We shall show that the set of all (weak) synchronizing triples for a given reduced LR-parser can be constructed with the use of generalized state transition functions.

Theorem 4.8.

(i) If $ST_1 = \{(p,Z,X) : \forall q \in \mathrm{NEXT}'(p,Z) :$
$\qquad (R(q,X) \vee \forall t \in \mathrm{NEXT}''(q,X) : R(t,X))\}$,
$ST_2 = \{(p,Z,X) : \forall q \in \mathrm{NEXT}''(p,Z) : (p,Z,X) \in ST_1\}$

then $ST = ST_1 \cup ST_2$.

(ii) If $WST_1 = \{(p,Z,X) : \exists q \in NEXT'(p,Z) :$
$(R(q,X) \vee \exists t \in NEXT''(q,X) : R(t,X))\}$,
$WST_2 = \{(p,Z,X) : \forall q \in NEXT''(p,Z) : (q,Z,X) \in WST_1\}$

then $WST = WST_1 \cup WST_2$.

Proof. (an outline).

(i) We have to prove that

$$(p,Z,X) \in ST_1 \cup ST_2 \Leftrightarrow (p,Z,X) \in ST.$$

If: Consider the following three cases:

1. $(p,Z,X) \in ST_1 \wedge (\forall q \in NEXT'(p,Z) : R(q,X))$.

By lemmas 2.4 and 4.3 we have:

$$\begin{aligned}&(\forall \alpha : next(q_0, \alpha) = p)\ (\exists q \in Q; \beta, \beta') : next(q_0, \beta') = q \wedge \\ &\quad (\forall w : ([\alpha], ZXw) \vdash^+ ([\beta'], Xw) \wedge R(q,X) \wedge \\ &\quad\quad ([\beta'], Xw) \vdash^+ ([\beta], w))\end{aligned}$$

which implies (by definition 4.1) that $(p,Z,X) \in ST$.

2. $(p,Z,X) \in ST_1 \wedge (\forall q \in NEXT'(p,Z) : (\forall t \in NEXT''(q,X) : R(t,X)))$.

By lemmas 2.4, 4.3 and 4.5 we have:

$$\begin{aligned}&(\forall \alpha : (next(q_0, \alpha) = p)\ (\exists q,t \in Q; \beta, \beta', \beta'') : next(q_0, \beta') = q \wedge \\ &\quad\quad next(q_0, \beta'') = t \wedge \\ &\quad (\forall w : ([\alpha], ZXw) \vdash^+ ([\beta'], Xw) \wedge next(q,X) \neq \emptyset \wedge \neg(read(q,X) \vee X \in N) \wedge \\ &\quad\quad ([\beta'], Xw) \vdash^+ ([\beta''], Xw) \wedge R(t,X) \wedge \\ &\quad\quad ([\beta''], Xw) \vdash^+ ([\beta], w))\end{aligned}$$

which implies (by definition 4.1) that $(p,Z,X) \in ST$.

3. $(p,Z,X) \in ST_2$ i.e. $\forall q \in NEXT''(p,Z) : (q,Z,X) \in ST_1$.

By lemma 4.5 we have:

$$\begin{aligned}&(\forall \alpha : next(q_0, \alpha) = p)\ (\exists q \in Q; \alpha') : next(q_0, \alpha') = q \wedge \\ &\quad (\forall w : ([\alpha], ZXW) \vdash^+ ([\alpha'], ZXw)) \wedge (q,Z,X) \in ST_1)\end{aligned}$$

which implies, by the cases (1) and (2), that $(p,Z,X)\epsilon ST$.

Only if: If $(p,Z,X)\epsilon ST$ then, from definitions 2.3 and 4.1, we have

$$
\begin{aligned}
&(\forall\alpha : next(q_0,\alpha)=p)\ (\exists p',q''\epsilon Q;\ \alpha',\beta,\beta',\beta'') :\\
&\quad next(q_0,\alpha')=p' \wedge next(q_0,\beta'')=q'' \wedge\\
&\quad (\forall w : ([\alpha],ZXw) \vdash^{*} ([\alpha'],ZXw) \wedge R(p',Z) \wedge\\
&\qquad ([\alpha'],ZXw) \vdash^{+} ([\beta'],Xw) \wedge\\
&\qquad ([\beta'],Xw) \vdash^{*} ([\beta''],Xw) \wedge R(q'',X) \wedge\\
&\qquad ([\beta''],Xw) \vdash^{+} ([\beta],w)).
\end{aligned}
$$

Consider the following three cases:

1. $\alpha=\alpha'$, $\beta'=\beta''$.

By lemmas 4.3 and 2.4 we have:

$$\forall q\epsilon NEXT'(p,Z) : R(q,X)$$

which implies that $(p,Z,X)\epsilon ST_1$.

2. $\alpha=\alpha'$, $\beta'\neq\beta''$.

By lemmas 4.3, 4.5 and 2.4 we have:

$$\forall q\epsilon NEXT''(p,Z) : (\forall t\epsilon NEXT''(q,X) : R(t,X))$$

which implies that $(p,Z,X)\epsilon ST_1$.

3. $\alpha\neq\alpha'$.

By lemma 4.5 $\forall p'\epsilon NEXT''(p,Z)$ we have case (1) or (2) for (p',Z,X), which implies that $(p,Z,X)\epsilon ST_2$.

(ii) The proof is similar to the proof outlined above.□

In the following the set WST of weak synchronizing triples will be represented by a mapping

$$r : Q\times(N\cup\Sigma\cup\{\perp\}) \rightarrow 2^{(N\cup\Sigma)}$$

such that

$$r(p,X)=\{Z : (p,Z,X)\epsilon WST\}$$

called a synchronizing table.

The notion introduced above will be illustrated by an example.

Example 1.

For the grammar with productions:

(1) $S \to$ head;V;B.	(6) $L \to I;L$
(2) $V \to$ **var** dcl	(7) $I \to$ **while** exp **do** I
(3) $V \to V$;dcl	(8) $I \to$ **if** exp **then** I
(4) $B \to$ **begin** L **end**	(9) $I \to$ assgn
(5) $L \to I$	(10) $I \to B$

which generates the skeleton of a typical programming language (head denotes a program header, dcl - declaration, exp - expression, assgn- assignment). The LR-parser decision table for a reduced SLR(1)-parser is given in Fig.1. In this table the symbol * denotes an entry (q,a) such that read(q,a) is true.

The synchronizing table r for the above grammar is given in Fig.2.

5. Utilization of sychronizing triples in error recovery

To discover the main problems concerned with the utilization of synchronizing triples in error recovery we shall extend an LR-parser from definition 2.3 with very simple but practically useful recovery scheme which reminisces the panic mode strategy [1,14]. Under this scheme a reduced LR-parser functions in two modes called the normal mode and the recovery mode. After the detecting of an error in the normal mode, the parser switches to the recovery mode. In this mode the input string is searched for a synchronizing symbol which matches some state on the parse stack. Subsequently a right correcting symbol is placed before synchronizing one. The parser can then resume its work for at least one step (before encountering any further errors).

A configuration of reduced LR-parser with the error recovery scheme is a triple $([\alpha],\beta,t)$, where α, β are as in definition 2.3, and

$t \in \{\mathbf{normal}, \mathbf{recov}\}$ represents a mode of parser work.

Definition 5.1. A relation $\vdash$ on configurations of a reduced LR-parser with error recovery scheme is defined as follows:

1. The normal mode.

$$\begin{array}{lll} \forall w : (q_0 \cdots q_n, Xw, \mathbf{normal}) & & \\ & \vdash (q_0 \cdots q_n q_{n+1}, w, \mathbf{normal}) & \text{if } next(q_n,X) = q_{n+1} ; \\ & \cdot & \cdot \\ & \cdot & \cdot \\ & \cdot & \cdot \\ & \vdash \mathbf{stop} & \text{if } next(q_n,X) = stop ; \\ & \vdash^{err} (q_0 \cdots q_n, Xw, \mathbf{recov}) & \text{if } ERR(q_n,X)^{5)} . \end{array}$$

2. The recovery mode.

$$\begin{array}{lll} \forall w : (q_0 \cdots q_n, Xw, \mathbf{normal}) & & \\ & \vdash (q_0 \cdots q_i, ZXw, \mathbf{normal}) & \text{if } Z \in r(q_i,X), \ 0 \le i \le n \ \wedge \\ & & (\forall j : i < j \le n : r(q_j,X) = \emptyset) ; \\ & \vdash (q_0 \cdots q_n, w, \mathbf{recov}) & \text{if } X \neq \bot . \end{array}$$

Let us notice that the above parser is nondeterministic in its recovery mode and that the final configuration $([S],\bot, \mathbf{normal})$ is always obtainable because $r(q_0, \bot) = \{S\}$.

A brief analysis of the error recovery scheme outlined above clearly displays its evident drawbacks. The first of these is that after recovery from a given error the parser may sometimes find another error on the same input symbol. If $(q_0 \cdots q_n, Xw, \mathbf{normal})$ is an error configuration and, after the recovery, we have a configuration $([\alpha],ZXw,\mathbf{normal})$ such that $next(q_0, \alpha) = q_i$, $0 \le i \le n$, $Z \in r(q_i,X)$ then it is possible that

$$(\exists q \in Q, \beta) : next(q_0, \beta) = q \ \wedge$$
$$([\alpha],ZXw,\mathbf{normal}) \vdash^{+} ([\beta],Xw,\mathbf{normal}) \ \wedge \ ERR(q,X).$$

If $(q,X) \in WST$ then another correction can be made, but otherwise it is obvious that the insertion of the symbol Z was the wrong error repair in the context (α,X).

5) The symbol $\vdash^{err}$ denotes that during a given transition an error is reported.

The simplest way to avoid such confusing corrections is the use of synchronizing triples instead of weak synchronizing triples. Unfortunately this solution is too restrictive to be used in practical compilers (for typical LR-grammars for programming languages, many important synchronizing triples belong to WST\ST) .

A better idea is to find the largest set $PWST \subset WST$ of proper weak synchronizing triples such that

$$(p,Z,X)\in PWST \Leftrightarrow (\forall\alpha : next(q_0, \alpha))\ \exists\beta, \gamma:$$
$$\forall w : ([\alpha],ZXw) \vdash^{+} ([\beta],Xw) \wedge (([\beta],Xw) \vdash^{+} ([\gamma],w) \vee$$
$$\exists Z' : (next(q_0, \beta),Z',X)\in PWST).$$

To develop an algorithm for the computation of a set PWST we define a dependence relation ρ on a set WST of weak synchronizing triples.

Definition 5.2. We say that a weak synchronizing triple (p,Z,X) depends upon a triple (p',Z',X') in the set WST, what we denote

$$(p,Z,X)\ \rho\ (p',Z',X') \Leftrightarrow X = X' \wedge (\forall\alpha : next(q_0, \alpha) = p\ \exists\beta :$$
$$(\forall w : ([\alpha],ZXw) \vdash^{+}([\beta],Xw) \wedge (\exists\gamma : ([\beta],Xw) \vdash^{+}([\gamma],w) \wedge$$
$$p = p' \wedge Z = Z' \vee next(q_0, \beta) = p' \wedge ERR(p',X)))).$$

Lemma 5.3.

$$(p,Z,X)\ \rho\ (p',Z',X') \Leftrightarrow (X = X' \wedge ((p = p' \wedge Z = Z' \wedge (p,Z,X)\in ST) \vee$$
$$(p'\in NEXT'(p,Z) \vee (\exists t\in NEXT''(p,Z) :$$
$$p'\in NEXT'(t,Z)) \wedge ERR(p',X)))).$$

Proof. The lemma follows directly from definitions 4.1 , 5.2 and lemmas 4.3 and 4.5 .□

Corollary. $\forall(p,Z,X)\in ST : (p,Z,X)\ \rho\ (p,Z,X)$.

We can now define a set PWST in terms of generalized state transition functions.

Definition 5.4. A set PWST of proper synchronizing triples is the largest subset of WST such that

$$\forall(p,Z,X)\in PWST\ \exists(p',Z',X')\in PWST :$$
$$(p,Z,X)\ \rho\ (p',Z',X').$$

A set PWST can be easily computed by the following algorithm.

Algorithm 3.

Input: Sets ST and WST.
Output: A set PWST $\subset$ WST.
Method:

PWST := ST ; DELTA := WST\ST;
do ($\exists(p,Z,X)\in$DELTA , $q\in Q$) : ($q\in$NEXT'(p,Z) $\vee$ ($\exists t\in$NEXT''(p,Z) $\wedge$
$q\in$NEXT'(t,Z))) $\wedge$ ERR(q,X) $\wedge$ $\neg\exists Z'$: $(q,Z',X)\in$PWST $\cup$ DELTA $\rightarrow$
DELTA := DELTA\\{(p,Z,X)}
od ;
PWST := PWST $\cup$ DELTA.

Theorem 5.5. Algorithm 3 properly computes a set PWST.

Proof. (an outline)
Algorithm 3 systematically removes from a set DELTA = WST\ST all such triples (p,Z,X) which do not depend upon others, until no further removals are possible.□

Example 2.

For the grammar from example 1 we have:

$$\text{WST}\backslash\text{PWST} = \{(q_9, \textbf{end}, \ ; \ ), (q_9, \textbf{end}, \textbf{end}), (q_{10}, \textbf{end}, \textbf{end})\}.$$

Another drawback of the outlined recovery method is that sometimes for a given synchronizing pair there are many correcting symbols but there is no simple method for choosing the best one.

A third drawback is that while searching for a first synchronizing pair, the parser may discard a large portion of the source program. Consequently some errors may be left undetected, or, even worse, the parser may accidentally join (with a correcting symbol) two arbitrary parts of a sequential form, subsequently causing many spurious errors.

The first of these two disadvantages may be eliminated simply by restricting the domain of a function r or, equivalently, by introducing a partial function

$$r' : Q \times (N \cup \Sigma \cup \{\perp\}) \rightarrow (N \cup \Sigma)$$

such that

$$r'(p,X) = Z \quad \text{if} \quad (p,Z,X)\in PWST \wedge \neg(\exists Z' : Z' \neq Z \wedge (p,Z',X)\in PWST) .$$

Unfortunately this restriction eliminates not only troublesome triples, requiring more then one symbol lookahead, such as (q_{13}, {**while, if**} , exp) in our example, but also triples such as (q_{13}, {I,B,assgn}, ;), where

$$\forall\alpha : next(q_0, \alpha) = q_{13}, \forall w : ([\alpha] , assgn;w) \vdash ([\alpha] , I;w) \wedge$$
$$([\alpha] , B;w) \vdash ([\alpha] , I;w).$$

Therefore this problem requires a more sophisticated solution.

Definition 5.6. We say that a synchronizing triple (p',Z',X') dominates over a triple (p,Z,X) in PWST or that

$$(p',Z',X') > (p,Z,X) \Leftrightarrow (p = p' \wedge X = X' \wedge \exists q\in Q : q = next(p,Z) \wedge$$
$$reduce(q,X) = \{Z'\to Z\} \wedge \neg(read(q,X) \vee X\in N) \vee$$
$$reduce(p,Z) = \{Z'\to Z\} \wedge (read(p,Z) \vee Z\in N)).$$

It is evident that in the case of the error recovery scheme outlined above we may eliminate from our considerations all such triples which are dominated by others.

Let

$$PWST' = \{(p,Z,X)\in PWST : \neg(\exists(p',Z',X')\in PWST :$$
$$(p',Z',X') > (p,Z,X))\}$$

be a set of all proper weak synchronizing triples which are not dominated by others. A partial function

$$r'' : Q \times (N \cup \Sigma \cup \{\perp\}) \to (N \cup \Sigma)$$

such that

$$r''(p,X) = Z \quad \text{if } (p,Z,X)\in PWST' \wedge \neg(\exists Z' : Z' \neq Z \wedge (p,Z',X)\in PWST')$$

is less restrictive than a function r' (i.e. $r' \subset r''$) and consequently is more useful in any recovery scheme.

The synchronizing table r'' for the grammar from example 1 is given in Fig. 3.

The second disadvantage mentioned above is more serious. Its satisfactory elimination requires the condensation of a right context of an error before the attempt to recover is made. In the literature such a procedure is known as a forward move.Originally it was proposed for simple precedence grammars and parsers [6,12] in which all strings are always syntactically reduced in the same manner (independent of a context). Later it was extended to the family of

LR-parsers [5,13,16], where the task of phrase reduction is complicated by the fact that a correct analysis of input already scanned is needed in order to continue parsing. To circumvent this difficulty some kind of parallel parsing from the point of error is used and context independent reductions in the lookahead string are performed. In the error recovery only that sequence of parser actions is considered which is common to all paths of the parallel parse [13].

Because the cost, in space and time, of such LR-parser forward moves is great, we propose a new simplified method for their organization. This method is based on a concept of start symbols.

Definition 5.7. Let $\Phi : (N \cup \Sigma \cup \{\perp\}) \to 2^Q$ be a mapping such that

$$\Phi(X) = Q \cap \left(\bigcup_{p \in Q} NEXT'(p,X) \right).$$

A symbol X is called a start symbol if $\exists q \in Q : \Phi(X) = \{q\} \wedge next(q,\perp) \neq stop$.

In the following we shall characterize the set of start symbols using a partial function

$$start : (N \cup \Sigma \cup \{\perp\}) \to Q$$

such that

$$start(X) = q \quad \text{if } \Phi(X) = \{q\} \wedge next(q,\perp) \neq stop.$$

6. The practical error recovering LR-parser

Any practical error recovery scheme should have the following characteristics [2,9,15,18,20]:

i) It should use, as a control for the parser, information which is algorithmically obtainable from the language description.

ii) It should never fail to continue parsing. The interval of text that escapes analysis in the course of attempting recovery from an error should be minimal.

iii) It should introduce only a minimal number of spurious errors into an otherwise syntactically correct text fragment as a consequence of its previous recovery actions.

iv) It should not have higher space or time complexity then does the parser.

v) It should not degrade the performance of the parser on error-free input.

We believe that our two-level SRC scheme, which is a phrase oriented extension of an LR-parser from definition 5.1, with a special procedure for the correcting of local errors, meets all of these requirements.

6.1 The Simple Recovery and Correction scheme

Under the SRC scheme a reduced LR-parser has three modes of operation, called the forward, the correction, and the recovery[6].

In the forward mode the parser crawls along the input string until it is accepted or until an error is detected or until an attempt is made to reduce a phrase across the error detection point.

In the correction mode the parser tries to correct an error by inserting, deleting or replacing a single grammar symbol using synchronizing triples. If no such change is possible, then, at the second level, recovery is attempted.

In the recovery mode the parser tries to recover from an error

i) by condensing the right context of the error during a forward move (if the lookahead symbol is a start symbol) and replacing the broken phrase by a proper nonterminal;

ii) by inserting a correcting symbol (if the lookahead symbol is a synchronizing one and there is a state on the parse stack which matches it);

iii) by removing states from the parse stack until such a state is reached in which the lookahead nonterminal symbol can be shifted;

iv) by deleting the lookahead symbol and repeating the recovery procedure for the next input symbol.

Now we will describe the SRC scheme in terms of LR-parser moves.

A configuration of a reduced LR-parser with the SRC scheme is a triple $([\alpha],\beta,t)$ where

$[\alpha] \in (Q \cup \{!\})^+$ - represents the contents of the stack;

$\beta \in (N \cup \Sigma)^*\perp$ - represents the input tape;

$t \in \{\textbf{forw}, \textbf{corr}, \textbf{recov}\}$ - represents a mode of parser operation.

The symbol $! \notin Q$ represents an error state which marks the position of an

6) To simplify the descripion of the SRC scheme, we have omitted the normal mode.

error in the stack.

To simplify the parser description we extend the functions next, reduce and read such that:

$$\begin{array}{ll} next(!,X) = start(X) & \forall X \in dom\ start\ ; \\ reduce(!,X) = \emptyset & \forall X\ ; \\ read(!,X) = \emptyset & \forall X\ . \end{array}$$

Definition 6.1. A relation $\vdash$ on configurations of the reduced LR-parser with the SRC scheme is defined as follows:

1. The forward mode:

$$\begin{array}{ll}
\multicolumn{2}{l}{\forall w : (q_0 \cdots q_n, Xw, \mathbf{forw})} \\
\quad \vdash (q_0 \cdots q_n q_{n+1}, w, \mathbf{forw}) & \text{if } next(q_n,X) = q_{n+1}\ ; \\
\quad \vdash (q_0 \cdots q_i, Aw, \mathbf{forw}) & \text{if } reduce(q_n,X) = \{A \to \alpha\}\ \wedge \\
 & (read(q_n,X) \vee X \in N)\ \wedge \\
 & (\forall j : i < j \le n : q_j \ne !)\ \wedge \\
 & (i = n-k \vee i > n-k \wedge q_i = !)\ ,\ k = |\alpha|-1\ ; \\
\quad \vdash (q_0 \cdots q_i, AXw, \mathbf{forw}) & \text{if } reduce(q_n,X) = \{A \to \alpha\}\ \wedge \\
 & \neg(read(q_n,X) \vee X \in N)\ \wedge \\
 & (\forall j : i < j \le n : q_j \ne !)\ \wedge \\
 & (i = n-k \vee i > n-k \wedge q_i = !)\ ,\ k = |\alpha|\ ; \\
\quad \vdash (q_0 \cdots q_n, Xw, \mathbf{recov}) & \text{if } q_n = !\ \wedge\ ERR(q_n,X)\ ; \\
\quad \vdash^{err} (q_0 \cdots q_n, Xw, \mathbf{corr}) & \text{if } q_n \ne !\ \wedge\ ERR(q_n,X)\ ; \\
\quad \vdash \mathbf{stop} & \text{if } next(q_n,X) = stop.
\end{array}$$

2. The correction mode:

$$\begin{array}{ll}
\multicolumn{2}{l}{\forall w : (q_0 \cdots q_n, XYw, \mathbf{corr})} \\
\quad \vdash (q_0 \cdots q_n, ZXYw, \mathbf{forw}) & \text{if } Z = r''(q_n,X)\ ; \\
\quad \vdash (q_0 \cdots q_n, ZYw, \mathbf{forw}) & \text{if } Z = r''(q_n,Y)\ ; \\
\quad \vdash (q_0 \cdots q_n, Yw, \mathbf{forw}) & \text{if } n > 0\ \wedge \\
 & (\exists Z : next(q_{n-1},Z) = q_n\ \wedge\ Z = r''(q_{n-1},Y))\ ; \\
\quad \vdash (q_0 \cdots q_n !, XYw, \mathbf{recov}) & \text{if true.}
\end{array}$$

3. The recovery mode:

$$
\begin{array}{lll}
\forall w : (q_0 \cdots q_n, Xw, \mathbf{recov}) & & \\
\quad \vdash (q_0 \cdots q_n, Xw, \mathbf{forw}) & & \text{if } X \in \mathrm{dom\,start}\ ; \\
\quad \vdash (q_0 \cdots q_i, ZXw, \mathbf{forw}) & & \text{if } Z = r''(q_i, X) \wedge \\
 & & \neg(\exists j : i < j \leq n : (q_j, X) \in \mathrm{dom}\, r'')\ ; \\
\quad \vdash (q_0 \cdots q_i, Xw, \mathbf{forw}) & & \text{if } X \in N \wedge R(q_i, X) \wedge \\
 & & \neg(\exists j : i < j \leq n : ((q_j, X) \in \mathrm{dom}\, r'' \\
 & & \vee R(q_j, X)) \wedge (q_i, X) \in \mathrm{dom}\, r'')\ ; \\
\quad \vdash (q_0 \cdots q_n, w, \mathbf{recov}) & & \text{if } X \neq \perp.
\end{array}
$$

6.2 Implementation of the SRC scheme

A reduced LR-parser with the SRC scheme is nondeterministic while working in the correction mode or in the recovery mode. (There may not be a unique action which leads to the correction or to the recovery). The simplest and most straighforward method for deterministic implementation of such a parser is to fix a sequence in which possible transitions are selected. This approach is fundamentally limited but it makes the SRC scheme efficient and is difficult to remove without using much more space and time for organizing trial parses.

Any error handling parser not only should detect an error but should also explain the recovery action. The SRC scheme gives us the possibility for introducing a simple method for the production of clear messages which describe the error repair or the recovery in terms of the source program or such well known programming language components as declarations, statements and expressions.

In the correction mode a message describing the single symbol change could have the following format:

" Z inserted before X "
or " X changed to Z "
or " X deleted ".

In the recovery mode the simplest recovery message has the format:

$$\text{" program text since line xxxx char yy changed to } \begin{vmatrix} X \\ Z \\ ZX \end{vmatrix} \text{ ",}$$

with the point of error recovery clearly marked.

As a useful addition to these messages the parts of the program omitted (not analyzed) can be designated by underlining.

The cost of implementation of the SRC scheme is low. A synchronizing table r'' is sparse; consequently, for its representation, we can use any sparse table compression technique. We suggest the use of the same technique as is used for the LR-parser decision table (in the implementation [10] - the table factorization method by Joliat).

The performance of the LR-parser with the SRC scheme is illustrated in Fig. 5. It presents a derivation tree constructed during the analysis of a string which does not belong to the language over the grammar from example 1. In that figure all symbols introduced by the SRC scheme are circled.

The parser has detected and corrected the following four errors:

E_1 : expected "exp"
correction: "assgn" changed to "exp"
E_2 : expected ";" or "**end**"
correction: ";" inserted before "assgn"
E_3 : expected "**begin**" , "**while**" , "**if**" or "assgn"
correction: "<instruction list>" inserted before "**end**"
E_4 : expected ";" or "**end**"
correction: "**end**" inserted before "." .

6.3 Semantic considerations

A syntactic error recovery method must allow not only the continuation of the syntax analysis of a program text, but the semantic analysis as well. This is particularly important for languages with large collections of data types, as for example, Pascal or Ada, for which most of the semantic errors can be discovered during the compilation.

In a classical compiler, in which the semantic analysis is performed concurrently with the syntactic analysis [1], the continuation of the semantic analysis after recovery from a syntactic error may be troublesome. The reason is that the forward move destroys the canonical derivation of the input and adversely affects the order in which semantic action occurs. In addition, some recovery actions, which erase a part of the parse stack, can leave semantic informations inconsistent. However, if an attribute grammar is used and the parser builds an abstract syntax tree, then the interaction of the error recovering parser and the module of the multipass semantic analysis will be simplified. This interaction needs only the designation that a correcting symbol has been introduced during the recovery, and the assignment to all its attributes of a standard value **error** which fulfils all semantic relations defined on these attributes.

7. Conclusions and suggestions for further work

In this paper we have presented a simple but powerful table driven scheme for the handling of syntactic errors in an optimized (reduced) LR-parser. The recovery control information is based on the well-defined concept of synchronizing triples, and is algorithmically obtainable from the LR-parser decision table. Consequently this scheme may be easily incorporated into any LR-parser generator (it works with SLR(1), LALR(1), and LR(1) parser construction techniques). It also meets typical software engineering requirements; primarily it does not degrade the performance of the parser on a correct program text and it gives the possibility of a very compact representation of recovery control tables.

The quality of error handling by the proposed scheme is rather high; it corrects most of the local and the global context errors, and even enables the recovery from such diverse programming errors as declarations or statements out of order and, for example, the use of the Algol-type syntax in a Pascal program text.

There are many possibilities for improving the quality of error handling by our method. The restriction to a fixed sequence of choosing admissible corrections and recoveries, which makes this scheme so efficient, is difficult to remove without using much more space and time for organizing trial parses. The trial parses (backtracking) give the possibility of using:

a. the established and more precise methods for LR-parser forward moves [5,13,16];

b. the function r instead of the more restrictive function r'';

c. the cost functions and semantic informations for selecting the best correcting symbol (in many cases [7] semantic informations are as good as the right context of the error in limiting the set of possible corrections).

Acknowledgments

I am very grateful to Professor David Gries of Cornell University for reading this manuscript and for suggesting many useful comments which improved the presentation significantly. I also thank Sara Clark for her editorial assistance.

References

1. Aho, A.V., Ullman J.D. Principles of Compiler Design. Addison-Wesley, Reading ,Mass. 1977.

2. Boullier, P. Automatic syntactic error recovery for LR-parsers. In: Preliminary Document (vol.2) of State of Art and Trends in Compilation. IRIA, Montpellier 1978, 239-252.

3. Ciesinger, J. A bibliography of error-handling. SIGPLAN Notices, vol.14,1(1979), 16-26.

4. DeRemer, F., Pennello, Th.J. Efficient computation of LALR(1) lookahead sets. SIGPLAN Notices, vol.14,8(1979), 176-187.

5. Druseikis, F.C., Riplay, G.D. Error recovery for simple LR(k) parsers. Proc. of the Annual ACM Conf.,1976, 396-400.

6. Graham, S.L., Rhodes, S.P. Practical syntactic error recovery. CACM,vol.14,11(1975), 639-650.

7. Graham, S.L., Haley, S.P., Joy, W.N. Practical LR error recovery. SIGPLAN Notices, vol.14,8(1979), 168-175.

8. James, L.R. A syntax directed error recovery method. Technical Report CSRG-13, Univ. of Toronto, 1972.

9. Krawczyk, T. Error correction by mutational grammars. Information Processing Letters, vol.11,1(1980), 9-15.

10. Krol, J., Rosek, J. Some problems of a SLR(1)-parser construction, (in Polish). Submitted for publication, Podstawy Sterowania.

11. Krol, J., Wyrostek, P. Syntax error recovery by means of synchronizing pairs, (in Polish). Podstawy Sterowania,vol.10,2(1980), 109-124.

12. Leinius, R.P. Error Detection and Recovevy for Syntax Directed Compiler Systems. Ph.D. Thesis, Univ. of Wisconsin, Madison, 1970.

13. Mickunas, M.D., Modry, J.A. Automatic error recovery for LR(k) parsers. CACM,vol.21,6(1978), 459-465.

14. Musinski, J.E. Lookahead recall error recovery for LALR parsers. SIGPLAN Notices,vol.12,6(1977), 48-60.

15. Pai, A.B., Kieburtz, R.B. Global context recovery: a new strategy for syntactic error recovery by table driven parsers. TOPLAS,vol.2,1(1980), 19-41.

16. Pennello, T.J., DeRemer, F. A forward move algorithm for LR error recovery. Conference Record of 5th Ann. ACM Symposium on Principles of Programming Languages, 1978, 241-254.

17. Poonen, G. Error recovery for LR(k) parsers. In: Information Processing 77. North Holland, Amsterdam 1977, 529-533.

18. Rohrich, J. Methods for the automatic construction of error correcting parsers. Acta Informatica,vol.13,2(1980), 115-139.

19. Sippu, S., Soisalon-Soininen, E. A scheme for LR(k) parsing with error recovery. Part II: Error recovery, Part III: Error correction. Int.J. Computer Math.,vol.8,2(1980), 107-119, vol.8,3(1980) 189-206.

20. Tai, K. Syntactic error correction in programming languages. IEEE Tr. on Software Engineering,vol.SE-4,5(1978), 414-425.

	head	;	.	var	dcl	begin	end	while	do	if	then	assgn	exp	$\perp$	S	V	B	L	I
q_0	q_2														q_1				
q_1														stop					
q_2		q_3																	
q_3				q_5												q_4			
q_4		q_6																	
q_5					R_2^*														
q_6					R_3^*	q_8											q_7		
q_7			R_1^*																
q_8						q_8		q_{11}		q_{12}		R_9^*					R_{10}	q_9	q_{10}
q_9							R_4^*												
q_{10}		q_{13}					R_5												
q_{11}													q_{14}						
q_{12}													q_{15}						
q_{13}						q_8		q_{11}		q_{12}		R_9^*					R_{10}	R_5	q_{10}
q_{14}									q_{16}										
q_{15}											q_{17}								
q_{16}						q_8		q_{11}		q_{12}		R_9^*					R_{10}		R_7
q_{17}						q_8		q_{11}		q_{12}		R_9^*					R_{10}		R_{10}

Fig. 1. The LR-parser decision table for a reduced SLR(1)-parser generated from the grammar G shown in the example 1.

	head	;	.	var	dcl	begin	end	while	do	if	then	assgn	exp	$\perp$	S	V	B	L	I
q_0		head												S					
q_1																			
q_2				;												;			
q_3		V			var														
q_4					;	;											;		
q_5		dcl																	
q_6		dcl	B			begin		begin		begin		begin					begin	begin	begin
q_7														•					
q_8		I B assgn				begin	I B L assgn	begin		begin		begin	while if				begin	begin	begin
q_9		end	end				end												
q_{10}		end	end			;		;		;		;					;	;	;
q_{11}									exp										
q_{12}											exp								
q_{13}		I B assgn				begin	I L B assgn	begin		begin		begin	while if				begin	begin	begin
q_{14}						do		do		do		do					do		do
q_{15}						then		then		then		then					then		then
q_{16}		I B assgn				begin	I B assgn	begin		begin		begin	while if				begin	begin	begin
q_{17}		I B assgn				begin	I B assgn	begin		begin		begin	while if				begin	begin	begin

Fig. 2. The synchronizing table r for the grammar G.
A blank entry denotes $r(q,X) = \emptyset$.

	head	;	.	var	dcl	begin	end	while	do	if	then	assgn	exp	⊥	S'	V	B	L	I
q_0		head												S					
q_1																			
q_2				;												;			
q_3		V			var														
q_4					;	;											;		
q_5		dcl																	
q_6		dcl	B			begin		begin		begin		begin					begin	begin	begin
q_7														•					
q_8		I				begin	L	begin		begin		begin					begin	begin	begin
q_9			end																
q_{10}			end			;		;		;		;					;	;	;
q_{11}									exp										
q_{12}											exp								
q_{13}		I				begin	L	begin		begin		begin					begin	begin	begin
q_{14}						do		do		do		do					do		do
q_{15}						then		then		then		then					then		then
q_{16}		I				begin	I	begin		begin		begin					begin	begin	begin
q_{17}		I				begin	I	begin		begin		begin					begin	begin	begin

Fig. 3. The synchronizing table r'' for the grammar G.
A blank entry denotes $(q,X)\notin \mathrm{dom}\, r''$.

X	head	var	dcl	begin	while	do	if	then	assgn	V	L	I
start(X)	q_2	q_5	q_4	q_8	q_{11}	q_{16}	q_{12}	q_{17}	q_{10}	q_4	q_8	q_{10}

Fig. 4. The partial function start for the grammar G.

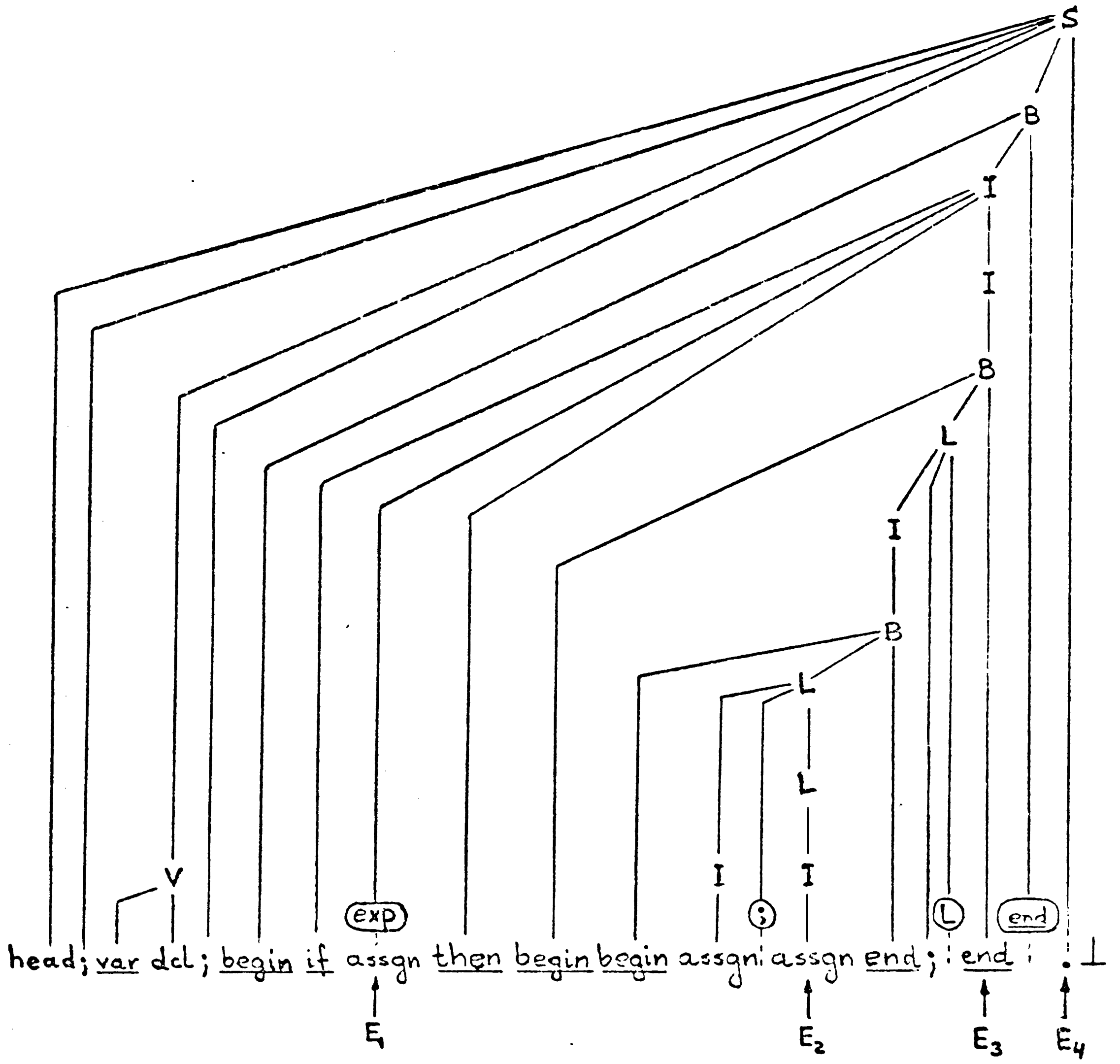

Fig. 5. A derivation tree illustrating the performance of the SRC scheme.