ON DEFINING ERROR RECOVERY IN CONTEXT-FREE PARSING

Seppo Sippu and Eljas Soisalon-Soininen

Department of Computer Science, University of Helsinki
Töölönkatu 11, SF-00100 Helsinki 10, Finland

1. INTRODUCTION

In compiler-compilers or parser generator systems it is necessary to have a part which generates programs for recovery from syntactic errors. This part should be designed so that it does not need much input other than the syntax of the language; otherwise a detailed knowledge of the implementation of the system may be needed for its successful use. In the case of bottom-up parsing, this design requirement for error recovery is fulfilled adequately in systems based on Leinius' idea of error isolation ([14]). Various applications of Leinius' idea are discussed in [1],[2],[3],[4],[15] and [16]; and a description of an implementation of this method in the case of LR parsing can be found in [11]. More refined versions of this basic idea developed recently are the systems of Graham and Rhodes ([7],[8],[12] and [17]) and Ciesinger [5]. Of these the former, however, is suitable only for precedence and bounded context parsing. A somewhat less automatic method, based on a different idea, is presented in Roehrich [18]. Another promising error recovery method, designed for LL(1) parsing, was recently introduced by Fischer et al. [6].

Most of these automatic or almost automatic recovery methods, however, have two disadvantages. First, their use (excepting only [6] and [18]) often leads to situations where several elements of the parse stack have to be discarded, which may considerably affect the later detection of semantic errors (see e.g. [6],[9],[10] and [18]). Second - which seems more serious to the authors - they are all highly dependent on the form of the context-free grammar in which the syntax is to be given (examples of this problem are presented in [11]). In other words, it is assumed that the user "programs" the recovery procedure for his compiler by using appropriate productions when writing the grammar. However, the user of a compiler-compiler has already to ensure that his grammar meets certain conditions in order to guarantee efficient parsing, while at the same time arranging that the semantics can be flexibly hanged on the grammar. Clearly it is undesirable that the user be faced with the additional requirement that the grammar should be proper for error recovery.

In this paper we first formalize the recovery method of Leinius [14] designed for LR parsing in terms of a generative model, and also define a nontrivial variation of it. We then give a contribution towards the solution of the problems mentioned above by

using a simple grammatical transformation that leads to refinement of recovery. In particular, we show by means of this transformation that certain types of replacement, deletion and insertion errors can be corrected.

2. PRELIMINARY DEFINITIONS AND NOTATION

We first review some basic notions of context-free grammars. If V is a set the elements of which are called characters or symbols, then V^0 denotes the set $\{\varepsilon\}$, where ε is the so called empty string, and V* denotes the set of all strings over V. If w is a string over V and k is a non-negative integer we stipulate that k:w is equal to the string formed by the first k symbols of w if the length of w is greater than k, and to w otherwise.

A quadruple G = (N,T,P,S) is a *(context-free) grammar* if N and T are finite disjoint sets, P a finite subset of N×(NUT)*, and S an element of N. Elements of the set N are called *nonterminals* and denoted by capital Latin letters A,B,...,Z. Elements of the set T are called *terminals* and denoted by small Latin letters from the beginning of the alphabet a,b,...,s. The elements (A,δ) of P are called *productions* and denoted by $A \to \delta$, where the nonterminal A is called the *left-hand side* of the production and the string δ the *right-hand side*. The symbol S is called the *start symbol*. We use the symbol V to denote the union of N and T. Terminal strings, i.e. strings over T, are denoted by small Latin letters from the end of the alphabet u,v,...,z, whereas small Greek letters α,β,γ and δ denote strings over V.

Let R be a relation on V*. A sequence $\alpha_1,\alpha_2,\ldots,\alpha_n$ of strings α_i over V is called an R *derivation*, or simply a *derivation*, from α_1 to α_n and written α_1 R* α_n if α_i R α_{i+1} for each i = 1,...,n-1 whenever n > 1. A derivation $\alpha_1,\ldots,\alpha_n$ is said to be *nontrivial*, written α_1 R⁺ α_n, if n > 1, and *direct*, written α_1 R α_n, if n = 2. A string β can be *obtained* by an R derivation from α if there exists an R derivation from α to β. If z is a terminal string then every R derivation from the start symbol S to z is called an R *parse* of z in the grammar G. The set of all terminal strings which can be obtained by R derivations from S is called the *language generated* by R and denoted by L(R). An algorithm that, given a terminal string z, determines an R parse of z if there is one and announces error otherwise, is called an R *parser* of the grammar G.

The *right(most)* relation on V*, denoted by $\underset{r}{\Rightarrow}$, is defined as follows: $\alpha A y \underset{r}{\Rightarrow} \alpha\delta y$ if $A \to \delta$ is a production in P and y is a terminal string. The language $L(\underset{r}{\Rightarrow})$ is denoted by L(G). A string γ is a *viable prefix* of G (e.g. [3]) if either γ = S or there is a derivation $S \underset{r}{\Rightarrow}^* \alpha A y \underset{r}{\Rightarrow} \alpha\beta\delta y$ such that γ = αβ. If γ is a string over V then FOLLOW(γ) denotes the set of strings 1:z such that z is a terminal string and there is a derivation $S \underset{r}{\Rightarrow}^* \gamma z$. In the following we shall assume that a grammar G has no *useless* nonterminals, i.e. for every nonterminal A there exists a derivation $S \underset{r}{\Rightarrow}^* \alpha A y \underset{r}{\Rightarrow}^* w$, where w is a terminal string.

3. EXTENDED PARSING

We first define a generative model by which the "recovery by reduction" error recovery method of Leinius [14] and variants of it can be described. We begin with the definition of the parser defined error in a terminal string ([16]). Let G be a grammar. If w = xy is a terminal string such that 1:y is not in FOLLOW(x), but L(G) nevertheless contains a string xu for some u, then we say that 1:y is the *parser defined error* in w with respect to L(G).

We now extend the definition of the rightmost relation $\underset{r}{\Rightarrow}$ so that strings with errors can be derived. Let γz be a string over V such that γ is a viable prefix of G and z is a terminal string. If 1:z is not in FOLLOW(γ) we say that the string γz is an *error sentential form* with the error point after γ. Further, if A is a nonterminal, then we say that a string αAy is in the *error relation* with $\alpha\beta xy$, an error sentential form with the error point after $\alpha\beta$, and write

$$\alpha Ay \underset{er}{\Rightarrow} \alpha\beta xy,$$

if αA is a viable prefix of G and 1:y is in FOLLOW(γA) for some γ over V. The pair $(A,\beta x)$, denoted by $A \rightarrow \beta x$, is said to be an *error production* which can be *used* in $\alpha\beta xy$. The right-hand side βx of the error production is called an *error phrase* in $\alpha\beta xy$, whereas the left-hand side A is called a *reduction goal* of βx in $\alpha\beta xy$. In addition, we say that the recovery can be *based* on βx and A in $\alpha\beta xy$. Finally, we define the *extended right(most)* relation on V*, denoted by $\underset{ex}{\Rightarrow}$, to be the union of the rightmost relation $\underset{r}{\Rightarrow}$ and the error relation $\underset{er}{\Rightarrow}$.

In order to serve as the basis of a generative model describing error recovery in right parsing the extended rightmost relation $\underset{ex}{\Rightarrow}$ should imply the basic property that $L(\underset{ex}{\Rightarrow}) = T^*$, i.e. every terminal string z has at least one extended right parse in G. If z is in L(G) then we can immediately conclude that there exists an extended right parse of z, since $\underset{r}{\Rightarrow}$ is a subrelation of $\underset{ex}{\Rightarrow}$. On the other hand, if z is not in L(G) it can be expressed as xy, where 1:y is the parser defined error in z. Since L(G) contains a string xu for some u, x must be obtained by a rightmost derivation from some viable prefix of G, say γ. Now, since 1:y is not in FOLLOW(x) and, consequently, not in FOLLOW(γ) either, we can conclude that there exists a derivation $S \underset{er}{\Rightarrow} \gamma y \underset{r}{\Rightarrow}^* xy = z$.

It would also be desirable that if a string z belongs to the language L(G) then every extended right parse of z in the grammar G would also be a right parse of z. Unfortunately this is not the case, as may be seen by considering the simple grammar with productions $S \rightarrow ab$, $S \rightarrow a$ and $S \rightarrow b$. There exists an extended right parse $S \underset{er}{\Rightarrow} Sb \underset{r}{\Rightarrow} ab$, although the string ab belongs to the language. If we wish to deny the possibility of this derivation, then the string Sb must not be obtained by an error derivation from S. This would imply that for the erroneous string bb the derivation $S \underset{er}{\Rightarrow} Sb \underset{r}{\Rightarrow} bb$ would not be possible, although in the authors' opinion it is the most natural. One reason why the latter derivation seems natural is that an LR parser which makes use of so

called default reductions, i.e. does not check the validity of reductions when only one reduce action is possible, would not detect the error in bb until after reducing the first b to S (see [1]-[4]).

However, it is interesting to note that if the grammar is LR(0) (see e.g. [1]-[4]) then every extended right parse of a string z in the language is also a right parse of z. In addition, it should be pointed out that the extended rightmost relation $\underset{ex}{\Rightarrow}$ can be defined so that this is the case for all LR(1) grammars. To do this the rightmost relation $\underset{r}{\Rightarrow}$ is replaced by a relation LR(1) defined as follows: αAy LR(1) $\alpha\delta y$ if $\alpha Ay \underset{r}{\Rightarrow} \alpha\delta y$ and 1:y is in FOLLOW(αA). The drawback of this definition is that it describes error recovery only in canonical LR parsing, whereas the original definition of $\underset{ex}{\Rightarrow}$ is capable to describe error recovery also for optimized LR parsers such as LALR and SLR parsers (see e.g. [1]-[4]).

To complete our model for error recovery in right parsing we state the following definition. An extended right parser of a grammar G is *valid*, if the extended right parse it determines for a string z is also a right parse of z whenever z belongs to the language L(G). To obtain a valid extended right parser of G we first construct a right parser of G and then augment it with a subrelation of the error relation $\underset{er}{\Rightarrow}$, called the *recovery operation* of an extended right parser and denoted by $\underset{rec}{\Rightarrow}$, which is one-to-one. That is, given an error sentential form γz with the error point after γ, there exists exactly one error phrase βx in γz and exactly one reduction goal A of βx in γz such that γ and z can be written as $\alpha\beta$ and xy, respectively, and $\alpha Ay \underset{rec}{\Rightarrow} \alpha\beta xy$.

4. ON DEFINING THE RECOVERY OPERATION

In this section we shall put forward a suggestion as to how the actual recovery operation of an extended right parser should be defined. For this purpose we shall first consider how to limit the often great number of different extended right parses a given terminal string may have, by restricting ourselves to certain subrelations of the error relation $\underset{er}{\Rightarrow}$ still without affecting the features essential for proper recovery.

In the following we use as an example the Algol-like grammar with productions

<program> → <block>
<program> → <compst>
<block> → begin <decllist> ; <stlist> end
<compst> → begin <stlist> end
<decllist> → <decllist> ; <decl>
<decllist> → <decl>
<decl> → integer i
<stlist> → <stlist> ; <st>
<stlist> → <st>
<st> → <block>
<st> → <compst>

<st> → s

<st> → ε

The erroneous string

begin integer ; integer i ; s end,

where 'i' is missing after the first 'integer', has, among many others, the following four extended right parses:

(1)
<program>
$\overset{*}{\underset{r}{\Rightarrow}}$ begin <stlist> end
$\underset{er}{\Rightarrow}$ begin integer ; integer i ; s end

(2)
<program>
$\overset{*}{\underset{r}{\Rightarrow}}$ begin <decllist> ; integer i ; s end
$\underset{r}{\Rightarrow}$ begin <decl> ; integer i ; s end
$\underset{er}{\Rightarrow}$ begin integer ; integer i ; s end

(3)
<program>
$\overset{*}{\underset{r}{\Rightarrow}}$ begin <stlist> ; <st> ; s end
$\underset{er}{\Rightarrow}$ begin <stlist> ; integer i ; s end
$\underset{er}{\Rightarrow}$ begin integer ; integer i ; s end

(4)
<program>
$\overset{*}{\underset{r}{\Rightarrow}}$ begin <decllist> ; integer i ; s end
$\underset{er}{\Rightarrow}$ begin integer ; integer i ; s end

The error productions used in the error sentential forms of the derivations (1), (2), (3) and (4) are, respectively, '<stlist> → integer ; integer i ; s', '<decl> → integer', '<st> → integer i' and '<stlist> → integer', and '<decllist> → integer'. Compared with derivation (3), the right-hand side of the error production used in the error sentential form of (1) seems unnecessarily long. Derivation (1) should therefore be ruled out. On the other hand, derivations (2) and (4) demonstrate the fact that the recovery based on <decl> does not essentially differ from that based on <decllist>. It is noteworthy that a similar observation can also be made as regards <block>, <compst>, <st> and <stlist>. The following definitions, however, provide natural conditions ensuring the elimination of such "unessential" reduction goals, and also the elimination of unnecessarily long error phrases ([14],[16]).

Suppose $\alpha\gamma\beta xuy$ is an error sentential form with the error point after $\alpha\gamma\beta$ such that both $\gamma\beta xu$ and βx are error phrases in $\alpha\gamma\beta xuy$. If either $\gamma \neq \varepsilon$ or $u \neq \varepsilon$ then we say that βx is *smaller than* $\gamma\beta xu$. Next suppose that A and B are reduction goals of an error phrase δ in an error sentential form $\alpha\delta y$. If B can be obtained by a nontrivial rightmost derivation from A, then we say that A is *more essential than* B with respect to δ in $\alpha\delta y$. In particular, the reduction goal A of δ in $\alpha\delta y$ is said to be *essential* if there exists no reduction goal B of δ in $\alpha\delta y$ more essential than A. We now combine these two concepts by defining a proper subrelation of the error relation $\underset{er}{\Rightarrow}$ as

follows: αAy is in the *least error* relation with αδy, written

$$\alpha Ay \underset{\text{ler}}{\Rightarrow} \alpha\delta y,$$

if A is an essential reduction goal of δ in αδy and there exists no error phrase in αδy smaller than δ. We call the union of the relations $\underset{r}{\Rightarrow}$ and $\underset{\text{ler}}{\Rightarrow}$ the *least extended right(most)* relation or simply the *least* relation.

The concept of a least derivation alone does not provide a sufficient criterion for uniquely selecting the error production to be used in a given error sentential form, even if the recovery were to be based on a fixed error phrase. For example, there exists exactly two least parses of the erroneous string above, namely (3) and (4), where the recovery is based on two different reduction goals (<stlist> and <decllist>, respectively) of the same error phrase (i.e. 'integer') in the same error sentential form. However, considering the "extraneous" error sentential form introduced in (3) by the "badly predicted" reduction goal <stlist>, it is obvious that (4) is more desirable and should be chosen in preference to (3). Thus there is no reason to rule out both of these derivations by resorting to the "unique essential replacement" criterion initially suggested by Leinius [14] and later used by Peterson [16], whereby an error production A → δ can be used only if A is a unique essential reduction goal of δ. (In fact, the extended right parse of the example erroneous string produced by this method would always be of the form (1)!) On the contrary, we can easily arrive at the desired result by defining yet another proper subrelation of the error relation $\underset{\text{er}}{\Rightarrow}$ in the following natural way:

Let αβxy be an error sentential form with the error point after αβ. First, we say that the parse of a nonterminal A is *incomplete* in αβxy with respect to β if αA is a viable prefix and there exists a terminal string z such that βz can be obtained by a nontrivial rightmost derivation from A. Now, let A be an essential reduction goal of βx in αβxy. We then say that αAy is in the *canonical error* relation with αβxy, and write

$$\alpha Ay \underset{\text{cer}}{\Rightarrow} \alpha\beta xy,$$

if either αAy = αβ = S or the parse of A is incomplete in αβxy with respect to β and, in addition to both these conditions, there exists no error phrase β'x' in αβxy smaller than βx with a reduction goal A' the parse of which is incomplete in αβxy with respect to β'. Correspondingly, we call the union of the relations $\underset{r}{\Rightarrow}$ and $\underset{\text{cer}}{\Rightarrow}$ the *canonical extended right(most)* relation or simply the *canonical* relation.

Clearly, derivation (4) is a canonical parse of the example erroneous string, and in this case even a unique such. In general, however, there may exist more than one canonical parse of a given terminal string. Consequently, when defining the actual recovery operation, a further strategy of specifying a choice ought to be introduced. When such a further selection decision is required, we could simply make the selection arbitrarily or we could try to achieve a better choice by somehow making use of the right context z of the error point in the error sentential form γz. We feel the

ultimate selection decision does not in most cases make any real difference in so far as the resulted recovery operation is still a subrelation of the canonical error relation $\underset{cer}{\Rightarrow}$.

A canonical extended right parser has been defined by Sippu [19]. The definition is an extension of the definition of the LR(1) parsing algorithm such that the extended right parser treats strings in the language in exactly the same way as the corresponding LR(1) parser. Some encouraging practical results have also been obtained for a subset of Algol containing 103 productions ([13]). The authors are preparing an English version of [19] in which the correspondence of canonical extended right parsing as defined in [19] and the relation $\underset{cer}{\Rightarrow}$ will be demonstrated.

5. ERROR CORRECTION BY EXTENDED PARSING

In the previous sections we have given a precise account of the basic recovery method of Leinius [14] for LR parsing and have indicated a nontrivial variation of it in the form of the canonical error relation. In this section we present a method for improving error recovery, and verify the improvement by using our formalization for error recovery and showing that the method leads to correction of certain kinds of errors whatever the form of the given grammar.

Although the error recovery methods based on the canonical error relation are theoretically attractive and concise, they still have the often harmful property that the quality of recovery is too sensitive to the form of the productions in the grammar. Already James [11] has noted that recovery may be unsatisfactory if the grammar has productions of the form $A \to \alpha b \beta$ where b is a terminal and β is a non-empty sequence of symbols containing at least one nonterminal. This drawback is clearly demonstrated by considering the grammar used as an example in the previous section. In fact, any string that does not begin with the symbol 'begin' can be obtained by a direct extended rightmost derivation only from <program>, <block> or <compst>, for instance

$$\text{<program>} \underset{cer}{\Rightarrow} \text{integer i ; s end}$$

is the only canonical parse of the string 'integer i ; s end' and

$$\text{<program>} \underset{cer}{\Rightarrow} \text{begni integer i ; s end}$$

is the only canonical parse of the string 'begni integer i ; s end', where the 'begni' stands for a misspelled 'begin'.

However, if instead of the production '<block> → begin <decllist> ; <stlist> end' our grammar had the productions '<block> → <begin><decllist> ; <stlist> end' and '<begin> → begin', then the respective canonical parses would be

$$\text{<program>} \underset{r}{\Rightarrow}^{*} \text{<begin> integer i ; s end} \underset{cer}{\Rightarrow} \text{integer i ; s end,}$$

where the error production used is '<begin> → ε', and

$$\langle program\rangle \overset{*}{\underset{r}{\Rightarrow}} \langle begin\rangle \text{ integer i ; s end}$$
$$\overset{}{\underset{cer}{\Rightarrow}} \text{begni integer i ; s end,}$$

where the error production used is '<begin> → begni'. These derivations can actually be considered as describing situations where the input string is corrected by adding the missing first terminal or, respectively, changing the erroneous first terminal. Consider on the other hand the erroneous string

i begin integer i ; s end,

which could be corrected by deleting the first 'i'. For obtaining an extended right parse that would describe this error and correction of it we cannot resort solely to the above substitution of the production '<block> → begin <decllist> ; <stlist> end'. Instead, the new production '<begin> → begin' could be split into the productions '<begin> → <ε>begin' and '<ε> → ε'. After this substitution the respective canonical parse would be

$$\langle program\rangle \overset{*}{\underset{r}{\Rightarrow}} \langle\varepsilon\rangle \text{ begin integer i ; s end}$$
$$\overset{}{\underset{cer}{\Rightarrow}} \text{i begin integer i ; s end,}$$

where the error production used is '<ε> → i'.

Altogether, the preceding examples suggest that extended parsing should be applied not to the given grammar but to a modified grammar defined as follows: Let G = (N,T, P,S) be a grammar. Given a string α in (NUT)*, we define $\hat{\alpha}$ to be the unique string which is obtained from α by replacing each terminal a in α by a new symbol <a> not in NUT. Let <ε> and S' be two more new symbols not in NUT. We now define the *augmented grammar* for G to be the grammar G' = (N',T,P',S'), where

$$N' = N \cup \{\langle a\rangle \mid a \text{ is in } T\} \cup \{\langle\varepsilon\rangle, S'\}$$

and

$$P' = \{A \to \hat{\alpha}\langle\varepsilon\rangle \mid A \to \alpha \text{ is in } P\}$$
$$\cup \{\langle a\rangle \to \langle\varepsilon\rangle a\langle\varepsilon\rangle \mid a \text{ is in } T\}$$
$$\cup \{\langle\varepsilon\rangle \to \varepsilon,\ S' \to S\langle\varepsilon\rangle\}$$

Trivially, L(G') = L(G) and right parses in the grammar G can be obtained from right parses in the augmented grammar G'. Moreover, it should be emphasized that this grammatical modification is given for conceptual purposes only and does not need to be carried out explicitly in practice. For example, any LR parser of the original grammar G constructed by the canonical, LALR or SLR construction algorithms (see e.g. [1]-[4]) can be interpreted as an optimized LR parser of the augmented grammar G' such that the reductions by new productions are combined with other actions as follows:

Action in the LR parser of the given grammar G	Combined actions in the LR parser of the augmented grammar G'
reduce by A → α	reduce by <ε> → ε reduce by A → $\hat{\alpha}$<ε>

shift a	reduce by <ε> → ε shift a reduce by <ε> → ε reduce by <a> → <ε>a<ε>
accept	reduce by <ε> → ε reduce by S' → S<ε> accept

Clearly, we are then able to construct the optimized extended right parser of the augmented grammar directly from the LR parser of the original grammar.

To characterize the usefulness of the given grammatical modification we need a few definitions. Let G be a grammar and z a terminal string. We say that a sequence $(e_1',e_1),\ldots,(e_n',e_n)$, where each e_i' and e_i is either a terminal or the empty string ε, *represents* the parser defined errors in z with respect to the language L(G) if z can be written as $z_0 = x_0e_1x_1\cdots e_nx_n$ such that each $1{:}e_{i+1}x_{i+1}\cdots e_nx_n$ is the parser defined error in the string $z_i = x_0e_1'x_1\cdots e_i'x_ie_{i+1}\cdots e_nx_n$ with respect to L(G). In particular, if $x_i \neq \varepsilon$ for each i = 1,...,n-1, we say that z has one parser defined error *locally*, and that the sequence $(e_1',e_1),\ldots,(e_n',e_n)$ represents the *isolated* parser defined errors in z.

The following theorem is an almost immediate consequence of the above definitions:

Theorem. Let G be a grammar. If a terminal string z has one parser defined error locally and the sequence $(e_1',e_1),\ldots,(e_n',e_n)$ represents the isolated parser defined errors in z with respect to the language L(G), then there exists in the augmented grammar G' a canonical parse of z the error productions of which are exactly $\langle e_1'\rangle \rightarrow e_1,\ldots,$ $\langle e_n'\rangle \rightarrow e_n$.

This theorem gives rise to the following definition: If Π is an extended right parse of a terminal string z in the augmented grammar G' such that every error production used in Π is of the form <e'> → e, where e' and e are terminals or the empty string ε, then we say that the unique string z' the right parse of which is obtained from Π by changing each error production <e'> → e to <e'> → e' is the *correction* of z defined by Π.

As an illustration consider the grammar for simple program structures given previously. The augmented grammar is then

<program'> → <program> <ε>
<program> → <block> <ε>
<program> → <compst> <ε>
<block> → <begin> <decllist> <;> <stlist> <end> <ε>
<compst> → <begin> <stlist> <end> <ε>
<decllist> → <decllist> <;> <decl> <ε>
<decllist> → <decl> <ε>

<decl> → <integer> <i> <ε>
<stlist> → <stlist> <;> <st> <ε>
<stlist> → <st> <ε>
<st> → <block> <ε>
<st> → <compst> <ε>
<st> → <s> <ε>
<st> → <ε>
<begin> → <ε> begin <ε>
<;> → <ε> ; <ε>
<end> → <ε> end <ε>
<integer> → <ε> integer <ε>
<i> → <ε> i <ε>
<s> → <ε> s <ε>
<ε> → ε

In the string

begin integer 2 ; s begin s end end end

it appears that '2' should be changed to 'i', a semicolon should be inserted between the first 's' and the second 'begin' and the final 'end' should be deleted. In the case of a typical optimized LR parser which makes use of default reductions (as mentioned in section 3) the following canonical parse of this erroneous string can be obtained:

<program'>
$\underset{r}{\Rightarrow}$ <program> <ε>
$\underset{cer}{\Rightarrow}$ <program> end
$\underset{r}{\Rightarrow}^*$ <begin> <decllist> <;> <stlist> <;> begin s end end end
$\underset{cer}{\Rightarrow}$ <begin> <decllist> <;> <stlist> begin s end end end
$\underset{r}{\Rightarrow}^*$ <begin> <integer> <i> ; s begin s end end end
$\underset{cer}{\Rightarrow}$ <begin> <integer> 2 ; s begin s end end end
$\underset{r}{\Rightarrow}^*$ begin integer 2 ; s begin s end end end

Here the error productions used are '<ε> → end', '<;> → ε' and '<i> → 2'. The correction of the erroneous string defined by this canonical parse is

begin integer i ; s ; begin s end end.

A system which generates an error recovery routine for any LALR(1) grammar (see e.g. [1]-[4]) is being prepared by one of the authors (Sippu). Augmented with this error recovery routine the LALR(1) parser of the grammar will be capable of producing a very near approximation to a canonical parse for every terminal string. It will also be possible to obtain a correction for most terminal strings that conform to the assumptions of the theorem stated above. Unfortunately the exact effect of the theorem will not always be achieved. This is mostly due to the decreased error detecting capability

of optimized LR parsers, especially those using default reductions.

Finally, it should be admitted that in a real compilation environment when an abstract parse tree (see e.g. [1]-[4]) is constructed alongside the parsing process, the parse tree corresponding to such an erroneous terminal string for which no correction could be obtained contains syntactically incorrect subtrees introduced by those error productions which are not of the form <e'> → e. Consequently the later semantic analysis may still be seriously affected (see again [6],[9],[10] and [18]). To remedy this shortcoming, the authors are working on the following idea (c.f. [6],[9] and [18]): given an error sentential form $\alpha\beta xy$ with the error point after $\alpha\beta$ such that $\alpha Ay \underset{cer}{\Rightarrow} \alpha\beta xy$, we could, instead of actually reducing by the error production $A \rightarrow \beta x$, select a string z_0 of "minimal cost" from the (by definition) non-empty set of terminal strings z for which there exists a nontrivial rightmost derivation $A \underset{r}{\Rightarrow}^{+} \beta z$ and replace x in $\alpha\beta xy$ by z_0, after which normal parsing could be resumed. Obviously, augmented with an error recovery routine working in this manner, the LR parser would be able to produce a syntactically correct parse tree for every terminal string.

ACKNOWLEDGEMENTS

The authors wish to thank Mikko Saarinen and Esko Ukkonen for pointing out some errors in earlier drafts of this paper. Thanks are also due to Professor Martti Tienari and to Jorma Sajaniemi for their comments.

This work was supported by a research project investigating translator writing systems. The project is led by Professor Martti Tienari and sponsored by the Academy of Finland.

REFERENCES

1. Aho,A.V.: Language theory in compiler design. In Yeh,R.T.(ed.): *Applied Computation Theory: Analysis, Design, Modeling.* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1976, pp. 185-249.
2. Aho,A.V. & S.C.Johnson: LR parsing. *Computing Surveys* 6:2 (1974), pp. 99-124.
3. Aho,A.V. & J.D.Ullman: *The Theory of Parsing, Translation, and Compiling. Vol.1: Parsing.* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1972.
4. Aho,A.V. & J.D.Ullman: *The Theory of Parsing, Translation, and Compiling. Vol.2: Compiling.* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1973.
5. Ciesinger,J.: Generating error recovery in a compiler generating system. In *GI - 4. Fachtagung über Programmiersprachen.* Springer-Verlag, Berlin - Heidelberg - New York, 1976, pp. 185-193.
6. Fischer,C.N., D.R.Milton & S.B.Quiring: An efficient insertion only error-corrector for LL(1) parsers. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages,* 1977, pp. 97-103.
7. Graham,S.L. & S.P.Rhodes: Practical syntactic error recovery in compilers. In *Conference Record of ACM Symposium on Principles of Programming Languages,* 1973, pp. 52-58.
8. Graham,S.L. & S.P.Rhodes: Practical syntactic error recovery. *Comm. ACM* 18:11 (1975), pp. 639-650.
9. Gries,D.: *Compiler Construction for Digital Computers.* John Wiley & Sons, New York, 1971.

10. Horning,J.J.: What the compiler should tell the user. In Bauer,F.L. & J.Eickel (eds): *Compiler Construction: An Advanced Course. Lecture Notes in Computer Science, Vol.21.* Springer-Verlag, Berlin - Heidelberg - New York, 1974, pp. 525-548.
11. James,L.R.: A syntax directed error recovery method. Technical Report CSRG-13, Computer Systems Research Group, University of Toronto, Toronto, May 1972.
12. Johns,C.B.: The generation of error recovering simple precedence parsers. Computer Science Technical Report No. 74/10, Department of Applied Mathematics, McMaster University, Hamilton, Ontario, July 1974.
13. Koskimies,K., K-J.Räihä,M.Saarinen,S.Sippu & E.Soisalon-Soininen: Definition and compiler of a subset of Algol (in Finnish), Internal Report, Series C, 1976/40, Department of Computer Science, University of Helsinki, Helsinki, May 1976.
14. Leinius,R.P.: Error detection and recovery for syntax directed compiler systems. Ph.D. Thesis, University of Wisconsin, Madison, 1970.
15. Lewis,P.M.,II, D.J.Rosenkrantz & R.E.Stearns: *Compiler Design Theory*. Addison-Wesley Publishing Company, Reading, Mass., 1976.
16. Peterson,T.G.: Syntax error detection, correction and recovery in parsers. Ph.D. Thesis, Stevens Institute of Technology, Hoboken, N.J., 1972.
17. Rhodes,S.P.: Practical syntactic error recovery for programming languages. Ph.D. Thesis, Technical Report 15, Department of Computer Science, University of California, Berkeley, June 1973.
18. Roehrich,J.: Syntax-error recovery in LR-parsers. In *GI - 4. Fachtagung über Programmiersprachen,* Springer-Verlag, Berlin - Heidelberg - New York, 1976, pp. 175-184.
19. Sippu,S.S.: Error recovery in LR parsing (in Finnish). Master's Thesis, Series C, 1976/22, Department of Computer Science, University of Helsinki, Helsinki, Feb. 1976.