Information Processing Letters 47 (1993) 189–196
Elsevier

27 September 1993

Syntactic error repair using repair patterns

In-Sig Yun, Kwang-Moo Choe and Taisook Han

Programming Languages Laboratory, Department of Computer Science, Korea Advanced Institute of Science and Technology, 373-1, Kusong-Dong, Yusung-Cu, Taejon 305-701, South Korea

Communicated by K. Ikeda
Received 22 September 1992
Revised 21 April 1993 and 24 June 1993

Abstract

Yun, I.-S., K.-M. Choe and T. Han, Syntactic error repair using repair patterns, Information Processing Letters 47 (1993) 189–196.

A syntactic error repair model is proposed, and is defined as a partial function from strings to sentences. The replacement of a substring of a string with a substring of a sentence is described by a *repair pattern,* which is roughly a pair of strings of grammar symbols. The model can be efficient with some restriction on repair patterns. An LR-based implementation of the model is discussed.

Keywords: Compilers; syntactic error recovery; syntactic error repair model; repair pattern

1. Introduction

Syntactic *ereror recovery* and *repair* schemes have been an important aspect of compiler design, and accordingly have received a great deal of attention in the literature (see, e.g., the bibliography by van den Bosch [10]).

Error repair schemes try to transform the erroneous substrings of a string into "similar" but syntactically correct substrings. In a more formal setting, if a strong $uxav$ is not in a language L such that u is a prefix of a sentence in L but uxa is not, then the schemes may attempt to find a pair of strings (x, y) such that uya is a prefix of a sentence in L, and repeat this process for $uyav$. We may consider finding such a pair of strings (x, y) as a main problem in local error repair. Usually, the pair (x, y) is found by some algorithm based on syntactic structures of grammars [1,4,5].

We present a method for describing such pairs (x, y). The pairs (x, y) are described by a *repair pattern,* which is roughly a pair (α, β) of two strings of grammar symbols such that $\alpha \Rightarrow^* x$ and $\beta \Rightarrow^* y$ with some restriction on β and y. The pair (α, β) is denoted by $\alpha \Rightarrow \beta$. We also propose an error repair model as a function using the repair pattern.

The following section describes the necessity for the repair pattern and reviews terminology and notations used. Section 3 contains a rationale of repair patterns and our error repair model. In Section 4, we formally define the repair pattern, and present an error repair model as a function. We also give theorems concerning some properties of the model and helping write repair patterns. Section 5 contains examples of repair patterns which can describe some kinds of errors that are difficult to repair in other schemes. This

Correspondence to: I.-S. Yun, Programming Languages Laboratory, Department of Computer Science, Korea Advanced Institute of Science and Technology, 373-1, Kusong-Dong, Yusung-Gu, Taejon 305-701, South Korea. Email: yunis@plhae.kaist.ac.kr.

demonstrates description ability of repair patterns. In Section 6, an algorithm for construction of an LALR parsing table with error entries is given, and an LR driver program for our model is discussed. The final section gives a conclusion. All proofs of theorems and lemmas in this paper are given in [11].

2. Motivation and terminology

Virtually all error repair schemes can have a poor repair for a substring x of a certain input string. If y is the most plausible repair for x, repair pattern $x \Rightarrow y$ requests the error repair scheme which supports repair pattern to handle this case. The model can transform a poor repair into a good one when a repair pattern is given for such poor case.

Since users of parser generators [3] may want trade-off between quality of repair and compile time, it is desirable that parser generators have switches for controlling quality of repair. Most parser generators produce error handlers which adopt an error scheme without such switches. In our scheme, because repair patterns are given as input to a parser generator, we can control the balance between efficiency and quality of repair.

Even the *minimum distance repair* [2] is not always the "most plausible" repair. Consider, for example, a Pascal fragment "**if** 1 < = i < = 7 **then** ...". A minimum distance repair seems "**if** 1 < = i + 7 **then** ...", but the most plausible repair is quite likely to be "**if** (1 < = i) **and** (i < = 7) **then** ...". Without mathematical knowledge that $1 \leqslant i \leqslant 7$ denotes $1 \leqslant i$ and $i \leqslant 7$, the repair cannot be obtained. The knowledge cannot be derived from the grammar, either. It, however, can be described by the repair pattern "E_1 < = **id** < = E_2 $\Rightarrow$ (E_1 < = **id**) **and** (**id** < = E_2)", where the subscripts in E_1 and E_2 distinguish two instances of E, which is a nonterminal for expression. A formal model based on only syntactic structure generated by a grammar does not know human errors.

In this paper we use the basic notions of grammars and parsing in [3]. A (*context-free*) *grammar* G is a quadruple (N, Σ, P, S), where N, Σ and P are finite sets of *nonterminals, terminals* and *productions,* respectively, and $S \in N$ is the *start symbol.* A symbol in $N \cup \Sigma$ is called a *grammar symbol.* We will use the following notational conventions. A, B and S denote nonterminals; X and Y denote grammar symbols; a, b and c denote terminals; u, v, w, x, y and z denote strings of terminals; and α, β, γ, δ and η denote strings of grammar symbols. A symbol X is said to be *useful* if either $X = S$ or $S \Rightarrow^* \alpha X \beta \Rightarrow^* w$ for some α, β and w. Otherwise X is *useless.* A grammar is said to be *reduced* if it contains no useless symbols. If L is a language and $uv \in L$, then u is called a *prefix* of L. $L(\gamma)$ denotes $\{w \mid \gamma \Rightarrow^* w\}$.

3. Rationale

This section deals with the rationale of the repair pattern and our error repair model. Let us start with a motivating example. A common error in **if** statement of Pascal is to put in an extraneous semicolon at the end of a statement between **then** and **else** [3,8]. To repair the error, the semicolon is to be deleted under "the context". We may express this type of error and its repair as

"**if** *expr* **then** $stmt_1$; **else** $stmt_2$

$\Rightarrow$ **if** *expr* **then** $stmt_1$ **else** $stmt_2$",

which is a repair pattern. The left and right part of $\Rightarrow$ are called the *pattern part* and *replacement part*, respectively. Here, **if, then, else** and ";" are terminals; and *expr* and *stmt* ($stmt_1$ and $stmt_2$ are two distinguished *stmt*'s) are nonterminals. An instance of this type error is "**if** $i > j$ **then** $max := i$; **else** $max := j$", and its repair is "**if** $i > j$ **then** $max := i$ **else** $max := j$".

Usually, as in the example, some terminals in repair patterns represent actual repairs, and the other terminals and all nonterminals represent a context. The pattern part can be any string of grammar symbols, but the replacement part cannot have new nonterminals which do not appear in the pattern part, since no context can be made.

When "**if** *expr* **then** $stmt_1$; **else** $stmt_2$" derived "**if** $i > j$ **then** $max := i$; **else** $max := j$", nonterminals *exp*, $stmt_1$ and $stmt_2$ derive "$i > j$",

"$max := i$" and "$max := j$", respectively. Such strings of terminals are obtained by function *yield*.

Under the context that the replacement part is allowed but its pattern part is not, if a portion of the input matches a string derived from the pattern part during parsing the input, then it is replaced with its repair, which is the replacement part with its nonterminals replaced by their *yield*'s. For example, the nonterminals *exp*, $stmt_1$ and $stmt_2$ are replaced with "$i > j$", "$max := i$" and "$max := j$", respectively. This is a rationale of the definition of function *repair*.

When a portion of the input matches to several pattern parts of repair patterns simultaneously, a repair pattern is chosen by comparing with "cost". For this purpose, the *cost part* of a repair patterns is introduced. If two or more portions of the input which begin at the same position match pattern parts of repair patterns, then it is reasonable to choose repair patterns corresponding to the shortest portion for implementation efficiency. Hence we define LCR_R this way. In case several portions of the input match pattern parts of repair patterns, the repair pattern corresponding to the leftmost portion is chosen. The reason is that most parsers scan the input from left to right. This reflects the definition of $repairer_R$.

4. Repair pattern and error repair model

This section deals with the relevant definitions of the repair pattern and our error repair model. There is a string α of grammar symbols of an unambiguous grammar such that some string in $L(\alpha)$ can have several rightmost derivations from α. For example, consider an unambiguous grammar for expressions from [3]:

$$(\{E, T, F\}, \{+, *, (,), \mathbf{id}\},$$
$$\{E \to E + T,\ E \to T,\ T \to T * F,\ T \to F,$$
$$F \to (E),\ F \to \mathbf{id}\},\ E).$$

Then the string **id** * **id** * **id** has two rightmost derivations from the string $E * T$. A string α of grammar symbols is said to be *ambiguous* if some strong in $L(\alpha)$ has two or more rightmost derivations from α. Otherwise, α is *unambiguous*. For string β of grammar symbols, N_β is defined to denote the set of all nonterminals which appear in β on the assumption that each symbol in β is distinguished.

Definition 4.1 (Repair pattern). Let α be an unambiguous string of a grammar G, β be a string in $(\Sigma \cup N_\alpha)^*$, and c be a nonnegative integer. Then (α, β, c) is called a *repair pattern* of G, and denoted by $\alpha \Rightarrow \beta(c)$ or by $\alpha \Rightarrow \beta$ if c is irrelevant. α, β and c are called the *pattern part*, *replacement part* and *cost part*, respectively.

When a string α derives a string x, $yield_\alpha(A, x)$ is defined to be x's substring derived from A. The first component of its domain is extended to strings naturally. The restriction that α is an unambiguous string makes $yield_\alpha$ a function.

Definition 4.2 (Function $yield_\alpha$). Let $\alpha = X_1 X_2 \ldots X_n$ be an unambiguous string of a grammar $G = (N, \Sigma, P, S)$. The function $yield_\alpha$ from $N_\alpha \times L(\alpha)$ to Σ^* is defined by $yield_\alpha(A, x) = x_i$ if $A = X_i$, where $X_j \Rightarrow^* x_j$ for all $1 \leqslant j \leqslant n$ with $x = x_1 x_2 \ldots x_n$. The domain of $yield_\alpha$ is extend to $(\Sigma \cup N_\alpha)^* \times L(\alpha)$ as follows:

$$yield_\alpha(\varepsilon, x) = \varepsilon,$$
$$yield_\alpha(a, x) = a \quad \text{for all } a \text{ in } \Sigma, \quad \text{and}$$
$$yield_\alpha(X\beta, x) = yield_\alpha(X, x)\, yield_\alpha(\beta, x).$$

The function $repair(u, v, \alpha \Rightarrow \beta)$ is defined to be the set of all triples $(x, y, \alpha \Rightarrow \beta)$ such that $x \in L(\alpha)$ is a prefix of v, y is $yield_\alpha(\beta, x)$, there is an error in x or at the symbol following x, and the error is repaired by the replacement of x with y.

Definition 4.3 (Function *repair*). Let R be a finite set of repair patterns of a grammar $G = (N, \Sigma, P, S)$. The function *repair* from $\Sigma^* \times \Sigma^* \times R$ to $2^{\Sigma^* \times \Sigma^* \times R}$ is defined by

$$repair(u, v, r) = \{(x, y, r) \mid r = \alpha \Rightarrow \beta,\ x \text{ is the longest}$$

prefix of v with $\alpha \Rightarrow^* x$, $y = yield_\alpha(\beta, x)$,
$v = xav'$, uxa is not a prefix of $L(G)$,
and $S \Rightarrow^* \gamma\beta\delta \Rightarrow^* uyaz$ for some z, δ,
and $\gamma \Rightarrow^* u\}$.

The phrase "the longest prefix of v" in above definition is used, since several prefixes of v can be derived from α. For example, prefixes **id** and **id** + **id** of a string **id** + **id** are derived from nonterminal E of the grammar for expressions.

Theorem 4.4. *Let $\alpha \Rightarrow \beta$ be a repair pattern of a grammar G. If $(x, y) \in repair(u, v, \alpha \Rightarrow \beta)$ then u is a prefix of $L(G)$, x is a prefix of v, and there exists an error in uxa and not in uya, where $v = xav'$.*

Any input string can be edited to a syntactically correct string by a sequence of primitive edit operations of inserting or deleting of a symbol, or replacing one symbol with another [1,2]. The repair patterns are used to describe the primitive edit operations, and may be classified according to them. We define that the repair patterns $\alpha\beta \Rightarrow \alpha Y\beta$, $\alpha X\beta \Rightarrow \alpha\beta$, and $\alpha Z\beta \Rightarrow \alpha W\beta$ are *for an insertion* of Y, *for a deletion* of X, and *for a replacement* of Z with W, respectively. We extend the definitions to two or more symbols, for example, $\alpha\beta \Rightarrow \alpha XY\beta$ is a repair pattern for insertions of X and Y.

The following theorem shows a limitation of the model, and implies that if any input should be repaired then some other repair scheme must be incorporated into the model in case all repair patterns fail.

Theorem 4.5. *For some grammar G, there is an error that cannot be described by any repair pattern in any finite set of repair patterns of G.*

The function $LCR_R(u, v)$ for a set R of repair patterns is defined to be the set of all triples in $\bigcup_{r \in R} repair(u, v, r)$ whose *cost* are the lowest. Note that the shorter the first component of a triple is, the lower its cost is.

Definition 4.6. (Function LCR_R, least-cost repair). Let R be a finite set of repair patterns of a grammar $G = (N, \Sigma, P, S)$. The function LCR_R from $\Sigma^* \times \Sigma^*$ to $2^{\Sigma^* \times \Sigma^* \times R}$ is defined by

$$LCR_R(u, v) = \{(x, y, r) \in A \mid cost(x, y, r) \leqslant cost(x', y', r') \text{ for all } (x', y', r') \in A\},$$

where $A = \bigcup_{r \in R} repair(u, v, r)$, $w = \max\{c \mid (x, y, \alpha \Rightarrow \beta(c)) \in A\} + 1$, and $cost(x, y, r) = |x| \times w +$ the cost part of r.

We are ready to present our error repair model as a function. First, let us imagine the syntactic error repair process performed by human beings. Suppose that there are a program with many syntactic errors and a compiler halting after the first detection of a syntactic error. One compiles the program with the compiler. When the compiler detects the first syntactic error, scanning left-to-right, it halts with an error message. Then one finds what is the error by the aid of the message, and repair the error in the program. One repeat this process for the repaired program until no syntactic errors are found by the compiler.

For a given set R of repair patterns, the function $repairer_R$ is recursively applied to the repair of the first syntactic error, if any. This is our error repair model. In the following, k denotes the point of the first error and $LCR_R(a_1 \ldots a_i, a_{i+1} \ldots a_n) \neq \emptyset$ implies that there is an error in $a_{i+1}a_{i+2} \ldots a_n$.

Definition 4.7 (Function $repairer_R$). Let R be a set of repair patterns of a grammar $G = (N, \Sigma, P, S)$. The function $repairer_R$ from Σ^* to Σ^* is defined by

$$repairer_R(a_1a_2 \ldots a_n) = \begin{cases} a_1a_2 \ldots a_n \\ \quad \text{if } LCR_R(a_1 \ldots a_i, a_{i+1} \ldots a_n) = \emptyset \\ \quad \text{for all } 0 \leqslant i \leqslant n, \\ repairer_R(a_1a_2 \ldots a_k y a_{k+|x|+1}a_{k+|x|+2} \ldots a_n) \\ \quad \text{otherwise,} \end{cases}$$

where $k = \min\{i \mid LCR_R(a_1 \ldots a_i, a_{i+1} \ldots a_n) \neq \emptyset\}$

and choose a (x, y, r) from $LCR(_R(a_1 \ldots a_k, a_{k+1} \ldots a_n)$ deterministically.

Theorem 4.8. *For any finite set R of repair patterns, the function repairer$_R$ is computable.*

We discuss how to write a repair pattern. Its pattern part is an unambiguous string, and its replacement part is a substring of a sentential form (otherwise, it is useless). For these, a few sufficient conditions are given below.

Theorem 4.9. *Let α and β be two unambiguous strings. Then so is $\alpha\beta$ if $\{y \mid xy \in L(\alpha)\}$ and $\{x \mid xy \in L(\beta)\}$ are disjoint.*

Theorem 4.10. *A substring of a sentential form of an unambiguous reduced grammar is unambiguous.*

Theorem 4.11. *Let G be a reduced grammar, and let γ be a substring of a string derived from any grammar symbols of G. Then γ is a substring of a sentential form.*

5. Examples of the repair patterns

The repair pattern may be used when the incorporated model has a poor repair for some error, and when we wish to guarantee correct treatment of rather common error. The following examples describe some kind of errors that are difficult to repair in other schemes with erroneous Pascal fragments. This demonstrates description ability of the repair patterns. The point of detection is indicated by "↑" and that of actual error guessed is indicated by "⇑".

Example 5.1. ⇑ **procedure** *factorial* (*x*: *integer*; **var** *fact*: *integer*): ↑ **integer**;

The **procedure** is used where a **function** seems to be intended. The difficulty of repair for this type is pointed out in [4]. A repair pattern "**procedure id** (*formalparms*): **typeid** ⇒ **function id** (*formalparms*): **typeid**" for replacement of **procedure** with **function** describes the error.

Example 5.2. **if** $A = 1$ **then** *write*(1) ⇑ ; ↑ **else** *write*(2);

The semicolon preceding the **else** is illegal and the obvious repair is to delete it. This is quite a difficult repair to realize because "**if** $A = 1$ **then** *write*(1)" reduces to a nonterminal before the error is detected [1,4,6]. A repair pattern "**if** *expr* **then** *stmt*; **else** ⇒ **if** *expr* **then** *stmt* **else**" for deletion of ";" describes the error.

Example 5.3. ...; ⇑ A ↑ $= B$ **then** *write*(1) **else** *write*(2); ...

An **if** is missed and obvious repair is to insert it at preceding A. Many error repairs have a great deal of difficulty with missing statement headers [7]. A repair pattern "*expr* **then** ⇒ **if** *expr* **then**" for insertion of **if** describes the error.

Example 5.4. $a := b$ ⇑ ↑ c ...

A symbol is missed between b and c. Three of many possible repairs are "$a := b + c$ ...", "$a := b$; c ..." and "$a := b$ [c ...". With additional input symbols which distinguish the current situation from the other possibilities, "$a := b\ c + \ldots$", "$a := b\ c := \ldots$", and "$a := b\ c]$ ...", more plausible repair, in each case can be chosen [1]. The three possible repairs are described by repair patterns "**id** := *expr* *expr* + ⇒ **id** := *expr* + *expr* +", "**id** := *expr* **id** := ⇒ **id** := *expr*; **id** := ", and "**id** := *exprlist*] ⇒ **id** := [*exprlist*]" for insertion of "+", ";", or "[".

Example 5.5. **if** ⇑ $a = b$ **or** c ⇑ d **then**...

An obvious repair is "**if** $(a = b)$ **or** $(c = d)$ **then** ...". Most repair schemes may not be entirely satisfactory because they give no insight into why the error was made. That is, expressions such as "... $a = b$ **or** $c = d$ ..." look correct and indeed are correct in many languages [7]. This kind of errors can be described by a repair pattern

"$sexpr_1$ **relop** $term_1$ **or** $sexpr_2$ **relop** $term_2$

⇒ ($sexpr_1$ **relop** $term_1$) **or** ($sexpr_2$ **relop** $term_2$)"

for insertions of "(", ")", "(" and ")", where the *sexpr* stands for the simple expression, and **relop** denotes the relational operators =, < >, <, >, <= or >=.

6. Discussion on LR-based implementation

This section discusses the possibility of an implementation of our error repair model with repair patterns $\alpha \Rightarrow \beta$ whose replacement part is a substring of a right-sentential form. Intuitively, we extend the LR-based parser to be able to parse α at states which "predict" β. When α is parsed successfully instead of β at one of the states, α in the parser stack is popped and β is parsed with the extended parser.

The domain of the function *goto* [3] is extended to strings: $goto(I_p, \varepsilon) = I_p$ and $goto(I_p, X\alpha) = goto(goto(I_p, X), \alpha)$, where I_p is a set of items. A definition of an efficient computation method for LALR(k) lookahead set LA_k can be found in [9]. The following theorem shows how to find the states which predict β.

Theorem 6.1. *Let β be a substring of a right-sentential form of an augmented grammar $G' = (N', \Sigma, P', S')$, and let I_p be a set of items in the collection of sets of* LALR(1) *items for G'. Let β' be the longest prefix of β such that $goto(I_p, \beta') = I_q \neq \emptyset$, and let $\beta = \beta' y$. If either $\beta = \beta'$ or $y \in LA_{|y|}(I_q, [A \to \delta \cdot])$ for some production $A \to \delta$, then there is γ such that $goto(I_0, \gamma) = I_p$ and $\gamma\beta w$ for some w is a right-sentential form.*

Step 2 of Algorithm 6.2, for each repair pattern $\alpha \Rightarrow \beta$, introduces item $[\beta \to \cdot\alpha, a]$ at states which predict β, where β in the item is treated as a new nonterminal. In step 3, sets of pairs of items are found. The function *closure* for LR(1) is found in [3]. Step 4 renames pairs of items. The last step constructs the parsing table, which has an additional type of move, repair E. The conflicts between ordinary item and "error" item are resolved in favor of the former. This cannot resolve all conflicts, even though a given grammar G is LALR(1).

Algorithm 6.2. Constructing an LALR parsing table with error repair entries.

Input. An augmented grammar $G' = (N', \Sigma, P', S')$ and a finite set R of repair patterns whose replacement part are a substring of some right-sentential form.

Output. An LALR parsing table with error repair entries.

Method.

1. Construct $C = \{I_1, I_1, \ldots, I_n\}$, the collection of sets of LALR(1) items for G'.
2. For each I_p in C, find $J_p = \{[\beta \to \cdot\alpha, a] \mid \alpha \Rightarrow \beta$ in R, β' is the longest prefix of β such that $goto(I_p, \beta') = I_q \neq \emptyset$, either $\beta = \beta'$ or $\eta \in LA_{|\eta|}(I_q, [A \to \delta \cdot])$ for some $A \to \delta$ in P' with $\beta = \beta'\eta\}$.
3. Find the smallest set D such that $D = \{(I_0, J_0)\} \cup \{(I, K) \mid (I_p, K_q) \in D$, X is a grammar symbol, $I = goto(I_p, X)$,

$$K = closure\Big(\big\{[\beta \to \alpha X \cdot \gamma, a] \mid [\beta \to \alpha \cdot X\gamma, a] \in K_q\big\} \cup J_p\Big)$$

and $(I, K) \neq (\emptyset, \emptyset)$.

4. Let $D = \{L_0, L_1, \ldots, L_m\}$ with $L_0 = (I_0, J_0)$.
5. State i of the parser is constructed from $L_i = (I_p, K_q)$. The tables *action* and *goto* for state i are determined as follows:
   (a) If $[A \to \alpha \cdot a\beta, b]$ is in $I_p \cup K_q$ and $goto(L_i, a) = L_j$, then set $action[i, a]$ to "shift j".
   (b) If $[A \to \alpha \cdot, a]$ is in $I_p \cup K_q$ with A in $N' - \{S'\}$, then set $action[i, a]$ to "reduce $A \to \alpha$".
   (c) If $E \neq \emptyset$, $[A \to \gamma \cdot a\delta, b]$ is not in I_p and $[A \to \delta \cdot, a]$ is not in I_p, then set $action[i, a]$ to "repair E", where $E = \{\alpha \Rightarrow \beta \mid [\beta \to \alpha \cdot, a] \in K_q\}$.
   (d) If $[S' \to S \cdot, \$]$ is in I_p, then set $action[i, a]$ to "accept".
   (e) If $goto(L_i, A) = L_j$, then $goto[i, A] = j$.
   (f) All entries not defined by rules (a) through (e) are made "error".

Let us consider the number of new states added by the above algorithm. Because the number of LALR(1) states is exponential in the size of the grammar at worst case [6], it is not difficult to conjecture that the number of these new states is also exponential. Lemma 6.3 shows that the conjecture is true. However, for typical programming language grammars, it is likely that the number of these new states is linear because the

number of LALR(1) states approximates to double of the number of nonterminals [6].

Lemma 6.3. *For each $n > 0$, let*

$$G_n = (\{S, A_0, B_0, A_1, B_1, \ldots, A_n, B_n\},$$
$$\{0, 1, a, a_0, a_1, \ldots, a_n\},$$
$$\{S \to A_0 B_0,\ A_0 \to a,\ A_n \to 1A_0a_n,$$
$$A_{i-1} \to 1A_i a_{i-1},\ A_i \to 0A_i a_i,$$
$$A_i \to 0A_0 a_i,\ B_0 \to a,\ B_n \to 1B_0 a_n,$$
$$B_{i-1} \to 1B_i a_{i-1},\ B_i \to 0B_i a_i,$$
$$B_i \to 0B_0 a_i,\ \text{where } 1 \leqslant i \leqslant n\},\ S)$$

be a grammar; and $R = \{A_0 \Rightarrow a_n,\ B_0 \Rightarrow a_n\}$. Then there is a constant $c > 0$ such that when Algorithm 6.2 is applied for the augmented grammar of G_n and the set R, the number of distinct second components of the set D is at least 2^{cn}.

Our LR parsing program (driver program) is similar to ordinary one [3]. When *action*$[s, a]$ is shift s', reduce $A \to \beta$, or accept, the program is the same with ordinary one. When *action*$[s, a]$ = repair E, it calls the **function** repairer with the current symbol a, the current parsing stack and the set E of repair patterns. If the **function** returns $(\alpha \Rightarrow \beta, T)$, then the message for "$\alpha$ is replaced with β" is printed, the parsing stack is replaced with T, and normal parsing is continued. If it returns fail or action $[s, a]$ = error, then some other error recovery schemes such as [5] are used.

```
function repairer(a, S, E)
begin
  for each repair pattern α ⇒ β in E, in low
  cost first order do begin
    T := S; {Copy the stack S into temporary
    stack T}
    pop |α| symbols off T;
    set ip to point to the first symbol of βa$
    and set shiftable to true;
    while the symbol pointed to by ip is not $
    and shiftable do begin
      let X be the symbol pointed to by ip;
      if action[top(T), X] = shift s or goto
      [top(T), X] = s for some s then {top(T)
      returns the top element of T}
        push s on T and advance ip to the
        next symbol
      else if action[s, a] = reduce A → γ,
      where X ⇒* aδ for some δ, and |T| >
      |γ| then
        pop |γ| symbols off T and then push
        goto[top(T), A] on T
      else shiftable := false
    end
    if shiftable then return (α ⇒ β, T)
  end
  return fail
end
```

Our model can be implemented efficiently when each replacement part of repair patterns is a substring of a right-sentential form and when Algorithm 6.2 does not cause additional parsing conflicts. If the total execution time of the **function** repairer during the parsing of the input is linear of the input length, the execution time of our parsing program is linear, since the ordinary LR parsing time is linear [6]. Each call of the **function** repairer takes a constant time, which is determined by the underlying grammar and repair patterns [11]. Since it guarantees a successful shift of symbol [11], the **function** is called at most the input length times. Hence its total execution time is linear.

7. Conclusion

We have formally defined the repair pattern, and given examples of repair patterns which can describe some kinds of errors that are difficult to repair in other schemes. This demonstrates expressive power of the repair patterns.

We have proposed a syntactic error repair model, using repair patterns, as a function from string to string. The model can be efficient when each replacement part of repair patterns is a substring of a right-sentential form and the repair patterns does not cause additional parsing conflicts. Furthermore, the model is considered to be flexible in the sense that it can control the trade-

off between quality and overhead of error repair (via repair patterns).

We have proposed an algorithm for constructing the LALR parsing table with error repair entries and LR parser driver using the parsing table.

Acknowledgment

The authors would like to thank the anonymous referees for valuable comments. Also, the coordinating efforts of the communicating editor, Professor K. Ikeda, should be deeply appreciated. The first author further wishes to thank Professor Do-Hyung Kim and Miss Jiyun Lee for valuable comments and proofreading the draft.

References

[1] S.O. Anderson, R.C. Backhouse, E.H. Bugge and C.P. Stirling, An assessment of locally least-cost error recovery, *Comput. J.* **26** (1) (1983) 15–24.

[2] A.V. Aho and T.G. Peterson, A minimum distance error-correcting parser for context-free languages, *SIAM J. Comput.* **1** (4) (1972) 305–312.

[3] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques, and Tools* (Addison-Wesley, Reading, MA, 1986).

[4] M.G. Burke and A.F. Gerald, A practical method for LR and LL syntactic error diagnosis and recovery, *ACM Trans. Programming Language Systems* **9** (2) (1987) 164–197.

[5] K.-M. Choe and C.-H. Chang, Efficient computation of the locally least-cost insertion string for the LR error repair, *Inform. Process. Lett.* **23** (6) (1986) 311–316.

[6] C.N. Fischer and R.J. LeBlanc, *Crafting a Compiler* (Benjamin/Cummings, Menlo Park, CA, 1986).

[7] C.N. Fischer and J. Mauney, On the role of error productions in syntactic error correction, Tech. Rept. #364, Dept. of Computer Sciences, University of Wisconsin-Madison, 1979.

[8] G.D. Ripley and C.D. Frederick, A statistical analysis of syntax errors, *Comput. Lang.* **3** (1978) 227–240.

[9] J.C.H. Park, K.-M. Choe and C.-H. Chang, A new analysis of LALR formalisms, *ACM Trans. Programming Language Systems* **7** (1) (1985) 159–175.

[10] P.N. van den Bosch, A bibliography on syntax error handling in context free languages, *ACM SIGPLAN Not.* **27** (1992) 77–86.

[11] I.S. Yun, Syntactic error repair using repair patterns, Dept. of Computer Science, *KAIST*, in preparation.