LEAST-COST SYNTACTIC ERROR REPAIR USING EXTENDED RIGHT CONTEXT

Jon Mauney

Computer Sciences Technical Report #495

February 1983

LEAST-COST SYNTACTIC ERROR REPAIR USING EXTENDED RIGHT CONTEXT

by

Jon Mauney

A thesis submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN -- MADISON

1983

LEAST-COST SYNTACTIC ERROR REPAIR

USING EXTENDED RIGHT CONTEXT

Jon Mauney

Under the supervision of Associate Professor Charles N. Fischer

We present an extension to the locally least-cost repair method. Our algorithm examines several symbols to the right of the error point, and computes a least-cost repair to that entire region. The algorithm uses a modifed context-free parser to consider all repairs in parallel; the parser used is that of Graham, Harrison, and Ruzzo. The repair algorithm is called only when an error is detected, and can be used in conjunction with LL(1) and LR(1) parsers.

The size of the repaired region can be dynamically controlled, expanding to include all relevant context. We discuss ways to determine whether the region is large enough, and present a simple, practical test. We also present results of an initial implementation of the algorithm, and measurements of typical region sizes.

Acknowledgements

I would like to thank my advisor, Charles Fischer, for invaluable assistance throughout the preparation of this thesis. Special thanks to Marvin Solomon and Raphael Finkel for their careful reading of my thesis in such a short time. I would also like to thank the other members of my examination committee, David DeWitt and James Smith.

Financial support for my graduate studies was provided by the University of Wisconsin, the National Science Foundation, Jack and Betsy Mauney, and Kathy Fox. Their assistance is gratefully acknowledged.

Table of Contents

Chapter 1 - Introduction and review of previous work

1.1. Introduction

The ability to handle errors is an important aspect of any human-computer interface. The handling of syntax errors by a compiler is an example of this problem, and one which has received considerable attention. Despite the efforts of many researchers over the years, the proper way to deal with syntax errors is still a subject of debate. This thesis represents yet another attempt to improve the state of the art.

1.2. Review of Previous Research

Since 1963, when Irons published the first automatic error-repair algorithm [20], numerous researchers have attacked the problem of handling syntax errors. The myriad schemes that have been presented may be compared according to several characteristics: the degree of automation, the goal of the method (repair versus recovery), the types of errors handled, the method of choosing among a set of possible actions, the amount of context around the error that is used, and whether there is a formal model of the results of the action taken.

We have divided error-handling strategies into three groups, *ad hoc* methods, automatic recovery methods, and automatic repair methods. However, the assignment of a technique to a category is not always clear-cut. Many methods comprise several parts, each of which falls into a different category. Typically, such a method will try a limited set of repairs, and fall back on a recovery algorithm if no repair is found.

1.2.1. Ad Hoc Methods

An obvious approach to error-handling is to add special routines to the parser, one to handle each error configuration that may arise. This approach has been used in production compilers [7]. The interface between parser and error-handlers may be accomplished by replacing the error entries in the parsing table by special entries that denote the recovery routine to be invoked [19].

Such *ad hoc* methods have the advantage that they perform very well on the common errors they were designed to anticipate. However, the compiler-writer must expend considerable effort to create these routines, and must renew that effort whenever the parser or underlying grammar is changed. Ad hoc methods do not necessarily perform well in situations that were not anticipated, and only careful testing can insure that no unpleasant actions, such as infinite loops, can occur.

Certain *ad hoc* methods may, however, be a useful addition to an automatic error-handling algorithm. Burke and Fisher [6] add "language specific maps" to their algorithm, which may encourage or prohibit specific repairs in specific situations. Fischer and Mauney [13] and Graham, Haley and Joy [16] advocate the addition of "error productions" to the grammar, in order to handle specific errors that are poorly handled by an automatic method. Such productions anticipate common errors and actually make them syntactically legal. When an error production is applied, however, it triggers a routine to announce the presence of the error. Error productions are especially useful for errors in which the point at which the error is detected differs from the point at which the error was made. In effect, the error production extends the amount of lookahead the parser uses.

Error productions are not restricted to *ad hoc* uses, but may also form the basis for automatic error-handlers. Aho and Peterson [1] repair errors by using a grammar extended with error productions for every possible error. Krawczyk's method [23] automatically adds error productions to the grammar, selecting error productions that handle a certain class of error, and that allow the resulting grammar to be parsed using an LR(1) parser.

1.2.2. Error Recovery Methods

Automatic error-handling strategies are often divided into the "error-recovery" methods and the "error-repair" methods, although the distinction is not always clear-cut. The aim of a recovery algorithm is to find some change to the state of the parser and the input stream that will allow parsing to continue. A repair algorithm, on the other hand, searches for some modification of the input stream that will turn it into a legal sentence of the language.

Many recovery methods are refinements of "panic mode" [2, page 391]. In panic mode, the recovery procedure skips ahead in the input until some "safe" symbol is found. The parse state is then adjusted so that the safe input symbol can be accepted, and parsing resumes. The "Global Context Recovery" of Pai and Kieburtz [29] scans for a member of a carefully selected set of "fiducial symbols," which have certain properties in the grammar that allow the algorithm to adjust the parse state reliably. A similar recovery method is "phrase-level recovery" [21,24]. In a phrase-level recovery, the algorithm attempts to find the smallest phrase containing the error, and to replace the corresponding input and stack symbols with the reduction goal of that phrase. Lewi *et al.* perform recovery similarly [26], using a table of "synchrotriples" to control the recovery.

Another family of recovery algorithms is headed by the work of Graham and Rhodes [18]. Before choosing a recovery action, Graham and Rhodes condense

the program around the point of error. Condensation consists of a "backwards move," in which additional reductions are made on the parse stack, and a "forward move," in which a portion of the remaining input is parsed and reduced to non-terminal symbols. The recovery algorithm then searches for actions that will link the condensed left and right contexts together into legal parsing configuration. These actions are less drastic than those in other recovery schemes. This method could be considered repair instead of recovery.

Graham and Rhodes implemented their idea for use with simple-precedence parsers. Several other researchers have applied the technique to LR parsing. It is much more difficult to do a forward move with an LR parser. Three different groups tackled the problem and came up with very similar solutions. Druseikis and Ripley [9] showed how a forward move could be accomplished and used it as their entire recovery algorithm; the remaining input was parsed by the forward move parser. Mickunas and Modry [28] and Pennello and DeRemer [30] used the forward move before choosing repairs.

1.2.3. Error Repair Methods

"Error repair" (or "error correction") algorithms search for a transformation of the input that will result in a syntactically correct sentence. Repair algorithms differ on several points: the kinds of transformations that will be attempted, the means by which they insure that the result is correct, the amount and kinds of information employed in choosing the repair, and the ability to repair all possible errors.

The transformations typically used by repair algorithms are insertion of a symbol or string of symbols, deletion of symbols, replacement of one symbol by another, and transposition of two adjacent symbols. A particular algorithm will employ some subset of these transformations. Most algorithms accommodate

insertion and deletion; few consider transposition.

There are two major methods for determining whether a transformation of the input can yield a correct sentence. One method is to make the transformation and then attempt to parse it; the other is to find a description of the legal continuations at the point of error, and match the remaining input to that description.

The earliest automatic error handling algorithm used the latter, continuation-matching method. Irons [20] used a special top-down parser, which simplified the generation of a continuation string. The algorithm then deleted input until it found a symbol that was part of the continuation string, and inserted any additional symbols necessary. Watt [37] adapted this idea for use with LL and LR parsers. Roehrich's [33] algorithm is similar; he augments the LR parser with a small table, containing one symbol per state of the parser, which allows the algorithm to find a continuation string. The continuation is, again, matched to the input.

Locally least-cost repair algorithms also proceed by continuation. The algorithms operate in a manner similar to Irons's algorithm, but the continuation string is a description of all possible continuations, and matching input to description must be more flexible, in order to choose the least-cost repair. Least-cost repair algorithms may be used with a variety of parsers, including LL(1) [14,15], LL(Regular) [31], LR(1) [8,11], recursive descent [4], and Earley's parser [3]. Least-cost repair will be discussed more fully in the next section.

The other method of determining whether a transformation yields a correct sentence is to make the change and attempt to parse it. This method is used by several repair algorithms. In order to reduce the number of transformations to be tested, most such algorithms will not consider insertion of a string of symbols, but

only of a single symbol (in addition to deletions, replacements and transpositions). Since the insertions are restricted, there may exist errors that the algorithm cannot repair; a second-level recovery algorithm is usually provided. These methods usually parse ahead over several input symbols, and weight the possible repairs according to the distance of the successful parse. Pai and Kieburtz [29] feel that parsing only two symbols is sufficient, and Graham, Haley, and Joy [16] parse a few symbols (about five). Feyock and Lazarus [10] and Burke and Fisher [6] parse much larger strings, up to 25 symbols [10]. Parsing ahead also allows the use of semantic routines to further weight the possible repairs; such checks are used in [6,10,16].

The repair algorithms mentioned so far are local repair algorithms; they require only that the parser be able to accept some number of additional symbols. Eventually, the entire string will be repaired in this piecewise manner. Other repair algorithms, however, take a broader view; they choose repairs so that the total change to the entire string is minimized.

Aho and Peterson [1] were the first to suggest a (globally) minimum-distance corrector. By augmenting the grammar with error-productions for all possible errors, they were able to consider all repairs in parallel, and choose the minimum distance repair. Since the extended grammar is ambiguous, they used Earley's parser. Lyon [27] followed a similar approach, but instead of augmenting the grammar, he extended the actions of the parser to have the same effect.

Whereas Aho and Peterson and Lyon used their extended parsers over the entire input string, Levy [25] suggested that minimum-distance repair be done only on the portion of the string that needs it. He discussed methods of backing up a parser to insure that the point of error is included in the repaired region, and suggested that the repaired portion only include as much of the string as

necessary to insure that the repair be optimal. These ideas are discussed in more detail in chapter 3. Levy showed that such repairs are theoretically possible, but he did not present an algorithm to find such repairs.

1.3. Locally Least-Cost Repair

This thesis builds upon the work of Fischer *et al.* [8, 11, 14, 15], who define a locally least-cost error repair model, using insertions and deletions and weights on those operations. Locally least-cost repair is defined as follows. The parser has accepted a prefix of the program, w, and detects a syntax error with the remaining input $a_1 \cdots a_n$. (w is a prefix of some sentence in the language, but wa_1 is not.) A locally least-cost repair is a minimum-cost combination of deletions of symbols $a_1 \cdots a_l$ and insertion of new symbols $b_1 \cdots b_m$, such that $wb_1 \cdots b_m a_{l+1}$ is a correct prefix. Backhouse [4] gives a similar definition, differing slightly in the handling of deletions, and in the use of replacements as well as insertions and deletions.

It is easy to see that any error situation has a local repair, although it could involve deleting all remaining input and inserting a completely different suffix. Therefore, a locally least-cost repair algorithm can handle any error, with using a secondary recovery algorithm.

Locally least-cost repair algorithms have several advantages. Foremost is the high-level model of the repairs. The actions of the algorithm can be understood and predicted without any knowledge of the implementation. The costs also provide a convenient handle for fine-tuning the algorithm, again without referring to the implementation. With properly adjusted costs, the quality of the repairs is very good in most cases.

There are cases, however, in which a locally least-cost repair algorithm does not perform well. One problem is that the point at which the parser detects an error may be to the right of the point at which the programmer actually erred. For example, in this fragment of Pascal code:

```
... ; A[i] = B[i] then ...
```

the error is not detected until the '=' is seen by the parser, since the 'A[i]' could be the beginning of an assignment statement. But the actual error is a missing keyword, if, before the 'A'. The if cannot be restored without backing up the parser. Backing up the parser is a technique used by some [6, 10, 25, 28], but it is rejected by many others, including Fischer *et al.*, because of the difficulty of determining when and how far to back up, and because backing up interferes with one-pass compilation.

The other problem with locally least-cost repair stems from the local nature of the definition. The locally least-cost algorithm uses only a small amount of information in choosing a repair (the parse stack, the costs, and one symbol of the remaining input). In some cases, this information is insufficient to yield the best repair. For example, in the Pascal program fragment:

```
... ; a := b c ...
```

the parser would announce error upon reading 'c', and the possible repairs would include:

```
... ; a := b + c ...
... ; a := b ; c ...
... ; a := b [ c ...
```

One of these repairs will have the lowest cost, and will always be chosen by the repair algorithm. (If several repairs have equally low cost, the algorithm must pick one. There is no advantage in trying to choose non-deterministically.) However, all of the repairs are plausible, and none can be the 'best' repair in all situations.

For example, if the error is that a '[' was omitted, and the repair is to insert ';', the parser will detect another error when the matching ']' is read. The announcement of two (or more) errors when only one exists is the defining symptom of a 'poor' repair.

Some source of additional information must be consulted, if the best repair is to be chosen. One source is the semantic information, such as the data type, associated with each symbol. Only repairs that are semantically valid will be considered. The use of semantic information has been investigated by Dion [8] and by Begwani [5].

Another source of information is the remainder of the input. If the algorithm looks ahead at additional input symbols, it may gather information that distinguishes the current situation from the other possibilities:

... ; a := b c ; ...

... ; a := b c := ...

... ; a := b c] ...

The algorithm can then choose the best repair in each case.

Several repair algorithms (notably those which choose repairs by trying and parsing) use additional lookahead [6, 16, 18, 29, 30]. However, those methods do not have a high-level model of the repairs, and they cannot handle all errors. Also, the usefulness of the lookahead may be affected by the presence of a second, nearby error, since it is difficult to distinguish spurious errors, caused by a poor repair, from nearby 'real' errors. Global repair algorithms [1, 27] can handle any error, and are not affected by clusters of errors, but they use Earley's parser over the entire input, and therefore are very slow.

We suggest a middle ground between locally and globally least-cost models; the repair algorithm will select some region of the sentence and find the least-

cost repair to that region that will allow the parse to continue through the region. Levy [25] and Tai [35] have described such a repair model as "locally least-cost", in contrast to globally least-cost models. Since we have previously considered locally least-cost repair to involve only a single symbol at a time [11, 14, 15], we will use the phrase "regionally least-cost" to describe the area between single-symbol and global models.

In the examples above, a regionally least-cost repair algorithm with a region size of two symbols would choose the repairs suggested (assuming a reasonable set of repair costs). For a more involved example, let us look at another portion of a Pascal program:

```
if i < j
    A[i := j
else A[j] := i;
```

The parser will announce an error upon encountering the first 'A', and the possible repairs include inserting **then** and inserting an arithmetic operator. Again, a well-tuned locally least-cost repair algorithm would choose the most likely repair, and would occasionally be wrong. A regionally least-cost algorithm with a region of three symbols would see "A[i", and would really be no better off than the locally least-cost algorithm. If the region is extended to four symbols, the algorithm must cope with a second error. The options now include inserting **then** before the "A" and "]" before the ":=", inserting an operator before "A" and replacing the "[" with **then**, and inserting an operator before "A" and replacing the ":=" with an operator. Extending the region further, to five or six symbols, shows that replacing the ":=" is undesirable, as it leads to still more errors. The choice, then, is a matter of where to insert the **then**, and it will be resolved by the costs of the other repairs (inserting "]" versus inserting an operator and replacing "["). We can also use semantic information (whether "A" is an array) to guide the choice. Begwani has

shown how semantic information can be used in a regional repair [6]. For this thesis, we will assume that no semantic information is used.

The previous example illustrates that repair algorithms that employ a forward move must deal with clusters of errors, and that the size of the region greatly affects the repair chosen. The definition of a regionally least-cost repair implies repair of any errors within the region. Tai's MCL(k) model [35] is an example of a regionally least-cost repair model, with a particular method of determining the region size. In Chapter 2 we present an algorithm that finds a regionally least-cost repair for any region, and in Chapter 3 we discuss ways of determining region size. In Chapter 4 we present empirical results of an initial implementation of the algorithm.

Chapter 2 - Regionally least-cost repair

In this chapter we define "regionally least-cost repair," which extends the least-cost repair model to strings of symbols. We present an algorithm that will find regionally least-cost repairs for any string, and that is usable in conjunction with LL and LR parsers. The problem of determining the length of the region to repair is addressed in the Chapter 3.

2.1. Notation and terminology

Throughout this thesis, unless otherwise stated, we will assume the prior choice of an arbitrary context-free grammar, $G=(V_n, V_t, S, P)$. We will also assume that G has been augmented with an end-marker. We will talk about the language generated by G, L(G), and about the prefixes of that language, $Pr(G) = \{ x \mid \exists y \text{ s.t. } xy \in L(G) \}$. The empty string is denoted by λ.

We will also assume a string of input symbols, $a_1 \cdots a_n$. Portions of the input string will usually be denoted by u, v, x, or y, and for specific substrings we will use the notation of [17]: w_i indicates the substring $a_1 \cdots a_i$, and $w_{i,j}$ indicates the substring $a_{i+1} \cdots a_j$. Catenation of strings will be indicated by juxtaposition; thus $w_i w_{i,j}$ is w_i catenated with $w_{i,j}$ (which happens to equal w_j).

The input typically will not be a sentence of L(G), due to the presence of one or more errors. It will be necessary to talk about "the error" in a portion of the input, meaning the difference between what was intended and what actually appears in the input. The repair algorithm cannot actually know what the error is, but the concept is useful for discussion of various situations. The "point of error" is the point where the difference occurs (an error with two points of difference

will be considered two errors). The "point of detection" is the point at which the parser detects that there is an error in the input and calls the repair algorithm. The two points do not necessarily coincide.

2.2. Definition of regionally least-cost repair

Our objective is to transform an arbitrary string, $x \in V_t^+$, into a string, y, such that $y \in L(G)$. The transformation will consist of a series of primitive edit operations, which are insertion, deletion, and replacement. (A fourth operation, transposition, is used by some other researchers, but is not considered here.)

In general, there may be more than one transformation and resulting string, y, which satisfy our primary goal. Therefore, we will search for a repair of least cost, based on cost functions for the three operations:

IC(a) gives the cost of inserting terminal symbol a
DC(a) gives the cost of deleting terminal symbol a
RC(a,b) gives the cost of replacing a with b

These costs are supplied with the grammar. We require that all costs be non-negative. The end-marker is assumed to be always correct, and therefore all edit operations involving the end-marker have infinite cost.

Replacements were not included in the repair models of [8,11,14] because of the difficulties that such operations apparently introduced. If arbitrary non-negative costs are allowed, a least-cost repair, even to a single-symbol error, may involve a large number of primitive operations. For example, given these costs:

IC(a) = 10 IC(b) = 10 IC(c) = 1
RC(a,b) = 1 RC(a,c) = 10 RC(b,c) = 1 RC(c,a) = 2

It is cheaper to replace a by b, and then replace that b by c (total cost=2) than to replace a directly by c (total cost=10). Similarly, it is cheaper to insert c and

replace it by a than to insert an a directly. Such sequences of operations can be more complicated, of course, and could run through every symbol in the alphabet. Consideration of these roundabout repairs would complicate the repair algorithm and increase its running time.

We have found, however, that such complications can be avoided. First, the indirect repairs described above can never be least-cost if the costs satisfy these "triangle inequalities":

For all $a,b,c \in V_t$

$$RC(a,b)+RC(b,c) \geq RC(a,c)$$
$$RC(a,b)+DC(b) \geq DC(a)$$
$$IC(a)+RC(a,b) \geq IC(b)$$
$$DC(a)+IC(b) \geq RC(a,b)$$

(The fourth inequality is not needed in practice, since that case is easily handled by the algorithms.)

Second, even if the costs do not satisfy the inequalities, the least-cost sequence of operations to obtain or delete a particular symbol is fixed. Therefore we can derive a new set of costs which satisfy the constraints. The new set of cost functions is equivalent to the original set, in that for any *least-cost* repair of x into y using the original costs, there is a repair of x into y with the same cost using the new cost functions. The repairs differ only in the intermediate steps used.

We find the new cost functions by building a weighted, directed graph of edit operations and searching for least-cost paths within the graph. In building the graph, we find it convenient to use Backhouse's [4] point of view: insertion is replacement of λ by a symbol and deletion is replacement of a symbol by λ.

Algorithm to "triangularize" cost functions:

1) Build a directed, weighted graph:

1.1) For each symbol $a \in V_t$, add a node labeled a.

1.2) Add a node labeled λ.

1.3) For each symbol $a \in V_t$, add an arc from node λ to node a with weight IC(a).

1.4) For each symbol $a \in V_t$, add an arc from node a to node λ with weight DC(a).

1.5) For each ordered pair $a,b \in V_t$, add an arc from node "a" to node b with weight RC(a,b).

2) Extract the new cost functions:

2.1) For each symbol $a \in V_t$, IC'(a) = length of shortest path from node λ to node a.

2.2) For each symbol $a \in V_t$, DC'(a) = length of shortest path from node a to node λ.

2.3) For each ordered pair $a,b \in V_t$, RC'(a,b) = length of shortest path from node a to node b.

Clearly, this procedure finds the least-cost series of operations to transform one symbol into another, or into the null string. In a least-cost repair, a higher-cost sequence with the same result will *never* be used, so we do not give anything up by adopting the derived cost functions.

Based on the (pre-processed) cost functions, we define two other functions that extend costs to nonterminals, strings and derivations:

$C(\lambda) = 0;$

$C(a_1 \cdots a_n) = IC(a_1) + \ldots + IC(a_n)$, for $a_i \in V_t$

$C(A) = \min \{ C(x) \mid x \in V_t^*, A \Rightarrow^* x \}$

$C(X_1 \cdots X_n) = C(X_1) + \ldots + C(X_n)$, for $X_i \in V$

For $X \in V$, $a \in V_t$, Derive(X,a) =

$$\min_{x,y \in V_t^*,\, b \in V_t} \left[\{\infty\} \cup \{ C(xy)+RC(a,b) \mid X \Rightarrow^* xby \} \right]$$

For $A,B \in V_n$, Derive(A,B) = $\min_{x,y \in V_t^*} \left[\{\infty\} \cup \{ C(xy) \mid A \Rightarrow^* xBy \} \right]$

In practice, the C and Derive functions will be pre-computed and stored in tables. Algorithms, not including replacements, are presented in [15]. It is a simple matter to add replacements to these algorithms. The cost functions can be triangularized when the tables are generated.

Replacements do not appear explicitly in the repair algorithm; they are handled implicitly by the Derive function. Derive considers not only the "normal" method of deriving a terminal a from a nonterminal X

$$X \Rightarrow^* xay$$

at cost C(xy), but also the possibility of changing the a into some b, then deriving b from X

$$X \Rightarrow^* xby$$

at cost RC(a,b)+C(xy). The assumption that RC(a,a)=0 allows us to combine these two cases in the definition.

One might expect that Derive(X,a), the cost of deriving a from X, should be the cost of deriving some symbol b from X plus the cost of replacing b by a. The proper definition, however, is the one given above. Our goal is not to produce an a starting from an X, but to turn the a into a string derivable from X. In this way we will repair the input into a sentence of the language.

Since replacements can be hidden in the Derive function, the only change needed to introduce replacements into the repair algorithms of [11, 14, 15] (besides the tables) is the ability to note that a replacement is called for and to make the change to the input string.

We are now ready to define a regionally least cost repair.

Definition. A *modification*, M, to a string, $x = a_1 \ldots a_n$, $a_i \in V_t$, is a series of edit

operations, $E_1E_2 \cdots E_nE_{n+1}$, where each E_i is a series of insertions, followed optionally by a delete or replace operation. The string resulting from the application of the modification, M, to a string, x, is written $M(x)$. Each E_i is applied to a_i, and consists of insertions *before* a_i, and possibly deletion or replacement of a_i itself. E_{n+1} consists only of insertions, which are made after a_n. The *cost of the modification*, $C(M(x))$, is the sum of the costs of the edit operations.

Definition. Given two strings $x, y \in V_t^*$, with x in Pr(G), a *repair of y following x*, is a modification, M, such that $xM(y)$ is in Pr(G).

Definition. A *regionally least-cost repair* of y following x is a repair, M, of y following x such that for any other such repair, N, of y following x, $C(N(x)) \geq C(M(x))$.

For bookkeeping purposes, our definition of modification and repair places insertions before an input symbol, with an extra edit position added to allow insertions at the end of the string. These trailing insertions are allowed by the definition because our algorithm uses such insertions in its intermediate stages. However, since costs are non-negative, such insertions cannot lower the cost of a modification, and aren't needed for least-cost repairs. The arguments presented in chapter 3 will be much simplified if trailing insertions are *not* allowed, so we will use the term *proper repair* to describe repairs which do not contain trailing repairs.

Our definition allows the deletion of the last symbol in the string. In this way, our definition is an extension of Backhouse's locally least-cost method [4] rather than Fischer's [8,11,14]. Fischer's definition requires that one more input symbol be accepted by the parser, whereas Backhouse requires only that the parser get beyond the error symbol. Although we feel that Fischer's definition is superior for

locally least-cost repair, Backhouse's is more appropriate for regional repair.

In the extreme case in which the region of repair is a single symbol, the regionally least-cost repair is the same as a locally least-cost repair (using Backhouse's definition). In the other extreme, a regionally least-cost repair of the entire string is the same as a globally least-cost repair.

2.3. An algorithm for globally least-cost repair

We will first develop an algorithm to find a least-cost repair to any string, then we will show how the algorithm can be applied to find a repair of any region within a string.

Aho and Peterson [1] perform their "least errors parse" by adding to the grammar error productions, which simulate modifications to the input. We will take a complementary approach, similar to that of Lyon [27], and use a modified parser on the original grammar. (This parser will form the basis of the repair algorithm, which will be called whenever an error is detected by a different parser, probably LL or LR. The second parser is responsible for analyzing the input in the absence of errors. The two parsers should not be confused.) The modified parser will be able to: (1) advance the parser state as if additional symbols were in the input (simulate insertions), (2) consume input without changing the parse state (simulate deletions), and (3) advance the parser state as if one symbol were in the input, while consuming some other symbol (simulate replacements). These changes to the parser are equivalent to the error productions used by Aho and Peterson, and render the grammar highly ambiguous; a general context-free parser is therefore necessary.

For our parser, we choose the algorithm of Graham, Harrison and Ruzzo [17]. This algorithm is related to Earley's parser, but is simpler and permits more efficient implementations, although the complexity is the same (see [17] for

details). The Graham-Harrison-Ruzzo parsing algorithm produces a triangular matrix, each cell of which contains a set of "dotted productions", $A \rightarrow \alpha \cdot \beta$, representing the possible parses of a corresponding substring of the input. The dot indicates that part of the production, α, has been used to match input symbols. The position of a cell in the matrix indicates the portion of the input covered; cell i, j covers symbols i+1 through j, inclusive. Cells that match a longer substring of the input are created by "pasting together" two existing cells; elements in cell i, j are found by pasting cell i, k to cell k, j for $i < k < j$. The parse is also advanced by pasting cells to the input symbols.

Informally, the Graham-Harrison-Ruzzo algorithm will move the dot if the symbol immediately to its right is matched. For example, if we have the dotted production $A \rightarrow \alpha \cdot B \beta$, and some production $B \rightarrow \gamma$ has been satisfied, then we can move the dot, yielding $A \rightarrow \alpha B \cdot \beta$. In order to accommodate λ-productions, the dot is also moved across symbols which can derive λ. The result is a set of dotted productions, each with the dot in a different location. The original pasting operators are:

For Q a set of dotted productions and R a set of symbols:

$$Q \times R = \{ A \rightarrow \alpha B \beta \cdot \gamma \mid A \rightarrow \alpha \cdot B \beta \gamma \in Q,\ \beta \Rightarrow^* \lambda, \text{ and } B \in R \}$$

$$Q^* R = \{ A \rightarrow \alpha B \beta \cdot \gamma \mid A \rightarrow \alpha \cdot B \beta \gamma \in Q,\ \beta \Rightarrow^* \lambda, \text{ and } B \Rightarrow^* C \text{ for some } C \in R \}$$

For Q and R sets of dotted productions:

$$Q \times R = \{ A \rightarrow \alpha B \beta \cdot \gamma \mid A \rightarrow \alpha \cdot B \beta \gamma \in Q,\ \beta \Rightarrow^* \lambda, \text{ and } B \rightarrow \delta \cdot \in R \}$$

$$Q^* R = \{ A \rightarrow \alpha B \beta \cdot \gamma \mid A \rightarrow \alpha \cdot B \beta \gamma \in Q,\ \beta \Rightarrow^* \lambda,\ B \Rightarrow^* C \text{ for some } C \in V_t^* \text{ and } C \rightarrow \delta \cdot \in R \}$$

The $\times$ and * products differ in that $\times$ considers only direct derivations, while * considers indirect derivations. The GHR algorithm also uses a function 'Predict' to set the stage for matches in the next column.

We introduce error-repair by extending these operations, and by attaching a running cost to each dotted production. We can simulate insertions into the program by moving the dot across any symbol, X, as if it derived λ, at a cost of C(X). This change to the parsing algorithm is equivalent to adding error productions to the grammar which allow all symbols to derive λ. Those symbols that derive λ in the original grammar will have C(X) = 0; in such cases, the action of the parser will be the same as it would in the original algorithm. In the * operator, we allow not only chain rules, but any derivation, $B \Rightarrow^* xDy$. Such a match is equivalent to inserting x and y, and has cost C(xy). The least-cost choice of x and y is given by Derive(B,D); in the case that $D \in V_t$, this extension also brings replacements into the repair algorithm. To simulate a deletion, we paste a dotted production to an input symbol, a, without moving the dot, at a cost of DC(a). Thus a symbol is consumed without advancing the parse state. We can now present our modified pasting operators.

If Q is a set of dotted productions and R is a set of symbols, then define

$$Q \times R = \{ A \rightarrow \alpha B \beta \cdot \gamma ; c \mid A \rightarrow \alpha \cdot B \beta \gamma ; c \in Q,\ B \in R,\ c = c + C(\beta) \}$$
$$\cup \{ A \rightarrow \alpha \cdot \beta ; c \mid A \rightarrow \alpha \cdot \beta ; c \in Q,\ c = c + DC(B),\ B \in R,\ B \in V_t \}$$

$$Q^* R = \{ A \rightarrow \alpha B \beta \cdot \gamma ; c \mid A \rightarrow \alpha \cdot B \beta \gamma ; c \in Q,\ \text{Derive}(B, D) \neq \infty,$$
$$D \in R,\ c = c + C(\beta) + \text{Derive}(B, D) \}$$
$$\cup \{ A \rightarrow \alpha \cdot \beta ; c \mid A \rightarrow \alpha \cdot \beta ; c \in Q,\ c = c + DC(B),\ B \in R,\ B \in V_t \}$$

If Q and R are sets of dotted productions, then define

$Q \times R = \{ A \rightarrow \alpha B \beta \cdot \gamma ; c \mid A \rightarrow \alpha \cdot B \beta \gamma ; c_1 \in Q, B \rightarrow \delta \cdot ; c_2 \in R, c = c_1 + c_2 + C(\beta) \}$.

$Q^* R = \{ A \rightarrow \alpha B \beta \cdot \gamma ; c \mid A \rightarrow \alpha \cdot B \beta \gamma ; c_1 \in Q, \text{Derive}(B, D) \neq \infty,$

$D \rightarrow \delta \cdot ; c_2 \in R, c = c_1 + c_2 + \text{Derive}(B, D) + C(\beta) \}$.

We must also modify the Predict function, which, in the original algorithm, sets up chain rules and insures that the contents of subsequent cells will legally follow the parse so far. In our algorithm, Predict sets up insertions as well. For R a subset of V_n:

$\text{PREDICT}(R) = \{ C \rightarrow \alpha \cdot \beta ; c \mid C \rightarrow \alpha \beta \in P, B \Rightarrow^* \gamma C \delta, B \in R, c = C(\alpha) \}$

For R a set of dotted productions,

$\text{PREDICT}(R) = \text{PREDICT}(\{B \mid A \rightarrow \alpha \cdot B \beta \in R\})$.

In the definitions of the pasting operators, we have separated the case in which R is a set of symbols from the case in which R is a set of dotted productions. The costs are computed slightly differently in the two cases. In the remainder of this thesis, however, it will be simpler to consider R a set of symbols *and* dotted productions, and the pasting operators to be a comnbination of the separate definitions given above.

The three functions contain all the modifications necessary to find the least-cost repair. The driving algorithm is unchanged.

Algorithm 2.0.

(* let $a_1 \cdots a_n$ be the n input symbols. The algorithm
will produce an (n+1)*(n+1) matrix t *)

```
1) begin
2)    t[0,0] := PREDICT({S});
3)    for j := 1 to n do
```



```
4)    begin (* build column j, given columns 0 through j-1 *)
5)       t[j-1,j] := t[j-1,j-1] * {a_j};
6)       for i := j-2 downto 0 do
7)       begin
8)          r := ( ∪ (t[i,k] × t[k,j])) ∪ t[i,j-1] × (t[j-1,j] ∪ {a_j});
                 i<k<j-1
9)          t[i,j] := r ∪ t[i,i]* r;
10)      end;
11)      t[j,j] := PREDICT( ∪ t[i,j]);
                           0≤i≤j-1
12)   end;
13) end.
```

The effect of these extensions to the parsing algorithm is the same as the effect of adding Aho and Peterson's error productions; the sets in the parse matrix are isomorphic to those that would be obtained using the modified grammar. Therefore, the correctness and complexity of the parser should be unchanged by the modifications. In fact, our proof of correctness follows much the sames lines as that in [34].

The only danger is the additional cost component of the dotted productions: since the cost is a non-negative integer, the presence of dotted productions that differ only in the cost could cause the size of a cell to be unbounded. However, it is easy to show that if two dotted productions in a set differ only in the cost component, then the one with higher cost can never participate in a least-cost repair; any parse can be made cheaper by using the other dotted production. Therefore, a higher cost duplicate can always be discarded, and the size of the cell is not affected by the presence of cost components.

Definition. A string $\alpha \in V_n^+$ *matches* a string $x \in V_t^*$ if there exists a modification M such that $\alpha \Rightarrow^* M(x)$. A dotted production, $A \rightarrow \alpha \cdot \beta ; c$, matches a string, x, if

there exists a modification, M, such that $\alpha \rightarrow M(x)$ and $C(M(x)) = c$. A set of dotted productions, or of symbols, matches a string if each element of the set matches the string.

In general, there will be a number of repairs that allow α to match x, and so there will be a number of dotted productions, $A \rightarrow \alpha \cdot \beta ; c$, that match x, all with different values for c. Since we are seeking a least-cost repair, we will be most interested in that dotted production with the lowest value for c. Therefore, we will use the phrases "the least-cost $A \rightarrow \alpha \cdot \beta ; c$" and "$A \rightarrow \alpha \cdot \beta ; c$ is least-cost" to indicate that c is the cost of the least-cost modification, M, such that $\alpha \Rightarrow^* M(x)$. These phrases do not imply anything about the least-cost repair of x, only that this is the least-cost way of using $A \rightarrow \alpha \cdot \beta$.

Definition. A dotted production, $A \rightarrow \alpha \cdot \beta ; c$, follows a string, u, if there exists a modification, M, such that $M(u)A \in Pr(G)$.

From these definitions, it is trivial for any dotted production to follow or match λ. In order for a dotted production to follow λ, we modify λ into whatever prefix is necessary to derive the left-hand side of the production from the start symbol (since the grammar is reduced, some such modification is always possible). To match λ with α, we modify λ into the least-cost string derivable from α. Thus, $A \rightarrow \alpha \cdot \beta ; C(\alpha)$ matches λ.

In matching λ, we have tacitly assumed that all symbols can be inserted and deleted with finite cost. If a user wishes to forbid the insertion (or deletion) of a symbol, he may assign it a cost of 'infinity'. Thus, some dotted productions may match or follow λ only with infinite cost. We then have the choice of continuing to claim that they match or follow, ignoring the distinction between finite and infinite, or of saying that, in fact they do not match or follow λ, and excluding them from the cells of the parse matrix. In the proofs that follow, we take the former course.

In practice, we would take the latter course; items with infinite cost would be immediately discarded for reasons of efficiency.

Lemma 2.1. If Q matches u and R matches v then $Q \times R$ and Q^*R match uv.

Proof. Since $Q \times R$ is a subset of Q^*R, we will prove the lemma for Q^*R. For every rule $A \rightarrow \alpha \cdot \beta ; c$ in Q^*R, there exist γ, δ, X such that $\alpha = \gamma X \delta$, $A \rightarrow \gamma \cdot X \delta \beta ; c_1 \in Q$, $X \rightarrow \sigma ; c_2$ (or just X) $\in$ R, and $c_1 + c_2 + C(\delta) = c$ (or $c_1 + Derive(X, v) + C(\delta) = c$). By assumption, these rules match u and v, so there exist repairs M_1 and M_2 such that $\gamma \Rightarrow^* M_1(u)$ and $X \Rightarrow^* M_2(v)$. Therefore $\alpha = \gamma X \delta \Rightarrow^* M_1(u) M_2(v) \delta$, so there is a modification of uv with cost c which allows $A \rightarrow \alpha \cdot \beta ; c$ to match uv. If R contains a terminal symbol, then $A \rightarrow \alpha \cdot \beta ; c$ may have gotten into Q^*R by a deletion of that symbol. In that case, there must be a $A \rightarrow \alpha \cdot \beta ; c$ in Q, with $c = \frown c\text{-}DC(v)$ (v must be the symbol in R which was deleted). $A \rightarrow \alpha \cdot \beta ; c$ matches u, so there is a modification, M, of u such that $\alpha \Rightarrow^* M(u)$. Then there is a modification, M, of uv, as follows: $M(uv) = M(u)$, and $C(M(uv)) = C(M(u)) + DC(v) = c$.

□

Lemma 2.2. If Q follows u then so do $Q \times R$ and Q^*R.

Proof. From the definitions of $\times$ and *, it is clear that the set of symbols on the left-hand sides of dotted productions in $Q \times R$ and Q^*R is a subset of the symbols on the left-hand sides in Q. By assumption, all those symbols follow u.

□

Lemma 2.3. If Q follows u and matches v, then Predict(Q) follows uv.

Proof. If $A \rightarrow \alpha \cdot \beta ; c$ is in Predict(Q) then there must be some $B \rightarrow \gamma \cdot C \delta ; c_1$ in Q, $C \Rightarrow^* yAz$, and there must be modifications such that $S \Rightarrow^* M_1(u) B \sigma \Rightarrow$

$M_1(u)\gamma C\,\delta\sigma \Rightarrow^* M_1(u)M_2(v)C\,\delta\sigma \Rightarrow^* M_1(u)M_2(v)y\,A\,z\,\delta\sigma$. Therefore, there is a modification, $M(uv) = M_1(u)M_2(v)y$, which allows $A \rightarrow \alpha\cdot\beta;c$ to follow uv.

Lemma 2.4. Every element of $t_{i,j}$ matches $w_{i,j}$ and follows w_i.

Proof. By induction which follows the order of computation of the algorithm.

Basis: $i=j=0$. A dotted production in $t_{0,0}$ must match λ and follow λ. As mentioned above, it is trivial for dotted productions to match and follow λ. The dotted productions created by Predict({S}) (line 2 of the algorithm) have the form $A \rightarrow \alpha\cdot\beta;C(\alpha)$, discussed above.

Induction: Case 1 ($0 \le i < j$). If $i=j-1$ then $t_{i,j}$ is computed on line 5 of the algorithm, otherwise it is computed as the union of Q×R and Q*R products on lines 8 and 9. In any case, each Q matches some $w_{i,k}$ by the induction hypothesis, and each R matches some $w_{k,j}$ by the induction hypothesis or by the fact that a_j matches $w_{j-1,j}$ ($=a_j$). Therefore, all the products match $w_{i,j}$ (Lemma 2.1) and the temporary variable r also matches $w_{i,j}$ (line 8). By hypothesis, $t_{i,i}$ matches $w_{i,i}$ and so $t_{i,i}{}^*r \cup r$ matches $w_{i,j}$ (line 9). Since each product has $t_{i,k}$ (for some k) on the left, and $t_{i,k}$ follows w_i by hypothesis, all the products must also follow w_i.

Case 2 ($0 < i = j$). The diagonal elements $t_{j,j}$ $j>0$ are set at line 11 to Predict($\bigcup_{i<j} t_{i,j}$) which, from the definition, equals $\bigcup_{i<j}$ Predict($t_{i,j}$). By the induction hypothesis, $t_{i,j}$ follows w_i (since $i<j$) and matches $w_{i,j}$, so by Lemma 2.3 Predict($t_{i,j}$) follows $w_i w_{i,j} = w_j$ and matches $\lambda = w_{j,j}$. Therefore $t_{j,j}$ follows w_j and matches $w_{j,j}$.

□

Lemma 2.5. If $A \rightarrow \alpha\cdot\beta;c_1$ matches $w_{i,j}$ and follows w_i then the least-cost $A \rightarrow \alpha\cdot\beta;c_2$ is in $t_{i,j}$.

Proof. Also by induction which follows the order of computation.

Basis: $i=j=0$. (Items must match and follow λ.) All dotted productions are represented in Predict({S}), and the cost of $A \rightarrow \alpha\cdot\beta;c$ in Predict({S}) is $C(\alpha)$, which by definition is the least-cost way to match λ.

Induction: Case 1 ($0 \le i < j$).

Suppose $A \rightarrow \alpha\cdot\beta;c$ follows w_i, matches $w_{i,j}$, and is least-cost. Then there are modifications M_1 and M_2 such that $S \Rightarrow^* M_1(w_i)A\,\delta \Rightarrow^* M_1(w_i)M_2(w_{i,j})\beta\delta$. Consider a particular derivation tree for this derivation of a least-cost repaired string. For simplicity, we will refer to this derivation as "the least-cost derivation." Let $\alpha = \alpha_1 X \alpha_2$ where "X" is the symbol which is an ancestor of a_j. Then α_1 matches $w_{i,k}$ and X matches $w_{k,j}$ for some $i \le k < j$. (See Figure 1.) Thus $A \rightarrow \alpha_1\cdot X\,\alpha_2\beta;c_1$ follows w_i and matches $w_{i,k}$, and the least-cost version must be in $t_{i,k}$ by the induction hypothesis. Now we must show that $A \rightarrow \alpha_1 X\,\alpha_2\cdot\beta;c$ is in one of the × or * products on lines 5, 8, or 9 of the algorithm. Having shown that some $A \rightarrow \alpha\cdot\beta;c$ is in $t_{i,j}$, we must show that it is least-cost. That proof is simple, since we choose the operands of × and * to be least-cost, and the operators themselves are defined to make least-cost choices. The only way to get a lower cost would be to match one of the input symbols from a different non-terminal symbol, but we are assuming that we are working with a least-cost derivation, so that is impossible.

Subcase a (all deletions). It may be that a_j is deleted. In that case $A \rightarrow \alpha\cdot\beta;c_1$, $c_1 = c - DC(a_j)$, matches $w_{i,j-1}$ and is least cost (if not, we could find a cheaper match for $w_{i,j}$). By the induction hypothesis, $A \rightarrow \alpha\cdot\beta;c_1$ is in $t_{i,j-1}$, and

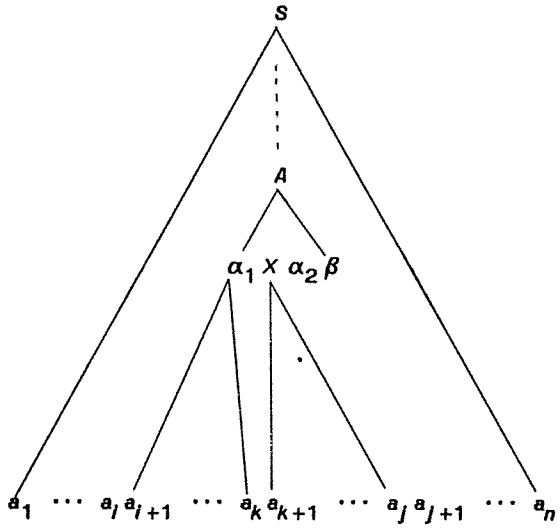

Figure 1: Lemma 2.5, case 1

$A \rightarrow \alpha \cdot \beta; c$ will be added to $t_{i,j}$ in line 5 if i=j-1, otherwise it will be added to r in line 8, and thus into $t_{i,j}$ in line 9.

Subcase b (i=j-1). If i=j-1 then i=k and $w_{i,j} = w_{j-1,j} = a_j$, so Derive(X,$a_j$) < ∞ and $A \rightarrow \alpha_1 X \alpha_2 \cdot \beta; c \in t_{j-1,j-1} * \{a_j\}$. Therefore it must be in $t_{j-1,j}$ (line 5 of the algorithm). Furthermore, $c = c_1 + \text{Derive}(X,a_j) + C(\alpha_2)$ must be least cost, since $A \rightarrow \alpha_1 \cdot X \alpha_2 \beta; c_1$ is least-cost and we are given that X is an ancestor of a_j in a least-cost derivation.

Subcase c (i<j-1). Let A' be the lowest common ancestor of a_{i+1} and a_j (these must be distinct input symbols, since i<j-1). Since A' is derived from A, A' follows

w_i. Also let $A' \rightarrow \alpha' \beta'$ be the rule used in the derivation, and $\alpha' = \alpha'_1 X' \alpha'_2$, with X' the ancestor of a_j. (See Figure 2.) X' is not the ancestor of a_{i+1}, for A' is the lowest common ancestor, so there is some k, i<k<j, and some modifications, M_1, M_2,

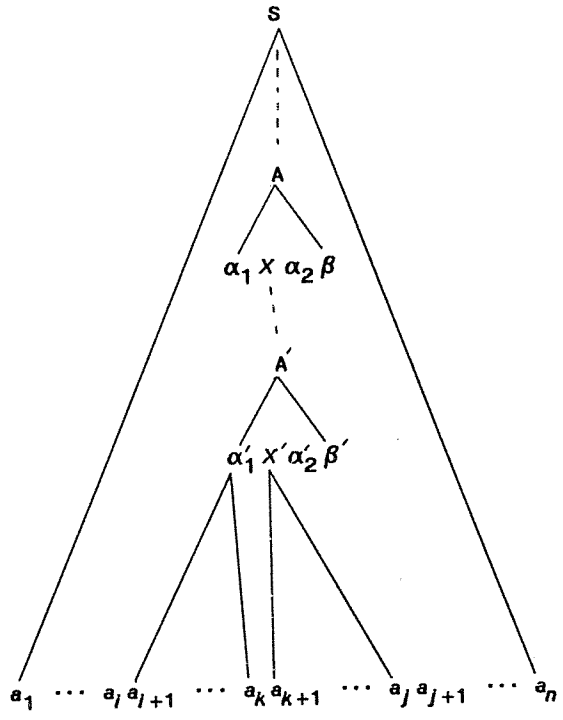

Figure 2: Lemma 2.5, case 1, subcase c.

such that $\alpha'_1 \Rightarrow^* M_1(w_{l,k})$ and $X' \Rightarrow^* M_2(w_{k,j})$. Thus, $A' \rightarrow \alpha'_1 \cdot X' \alpha'_2 \beta'; c'_1$ follows w_l and matches $w_{i,k}$ for some c'_1, and the least-cost such dotted production is in $t_{i,k}$, by the induction hypothesis. If $X' \in V_t$, then $X' = a_j$, $k = j-1$, and $A' \rightarrow \alpha' \cdot \beta'; c'$ is in $t_{i,j-1} \times \{a_j\}$. If $X' \in V_n$, suppose $X' \rightarrow \sigma$ is the first step in the derivation $X' \Rightarrow^* M_2(w_{k,j})$. Then the least-cost $X' \rightarrow \sigma \cdot; c'_2$ matches $w_{k,j}$ and follows w_k, so it is in $t_{k,j}$, and $A' \rightarrow \alpha' \cdot \beta'; c'$ will be in $t_{i,k} \times t_{k,j}$. Thus, whether X' is in V_t or V_n, $A' \rightarrow \alpha' \cdot \beta'; c'$ will be added to variable r by line 8 of the algorithm. Furthermore, c' must be the lowest cost for $A' \rightarrow \alpha' \cdot \beta'$, given the definition of the $\times$ product, the fact that c'_1 and c'_2 are least-cost, and the fact that X' is an ancestor of a_j in the least-cost derivation. If $A' = A$, then $\alpha' = \alpha$, $\beta' = \beta$, and $c' = c$, so $A \rightarrow \alpha \cdot \beta; c$ will be in $t_{i,j}$ (and will be least-cost). Otherwise, by our choice of A, it must be true that $X \Rightarrow^* uA'v$, that is, $k = i$ and $A \rightarrow \alpha_1 \cdot X \alpha_2 \beta; c_1$ is in $t_{i,l}$ $(= t_{i,k})$. Since $A' \rightarrow \alpha' \cdot \beta'; c'$ is in r, so is $A' \rightarrow \alpha' \beta' \cdot; c''$ (since the dot continues to move across all symbols), so $A \rightarrow \alpha \cdot \beta; c$ will be in $t_{i,i} * r$ and will be added to $t_{i,j}$ in line 9 of the algorithm. We know that $A' \rightarrow \alpha' \cdot \beta'; c'$ is least-cost and that β' does not contain any ancestors of input symbols, so $A' \rightarrow \alpha' \beta' \cdot; c''$, $c'' = c' + C(\beta')$, must be least-cost. Therefore, given that $A \rightarrow \alpha_1 \cdot X \alpha_2 \beta; c_1$ is least-cost and X is an ancestor of A' in the least-cost derivation, $A \rightarrow \alpha \cdot \beta; c$, $c = c_1 + \text{Derive}(X, A') + c''$, must also be least-cost.

Case 2 ($i = j$) Let $C \rightarrow \gamma \cdot \delta; c$ follow w_j and match λ. Then $S \Rightarrow^* M(w_j) C \sigma$. Again, consider the derivation tree for this derivation. Let A be the lowest common ancestor of C and a_j, and suppose the rule is $A \rightarrow \alpha B \beta$, where α contains an ancestor of a_j and B is an ancestor of C. Let $l < j$ be such that a_{l+1} is the leftmost input symbol derived matched by A. Then $A \rightarrow \alpha \cdot B \beta; c$ matches $w_{l,j}$ and follows w_l. By induction hypothesis, the least-cost $A \rightarrow \alpha \cdot B \beta; c$ is in $t_{i,j}$. Since B is an ancestor of C, we know that $B \Rightarrow^* uCv$, so $C \rightarrow \gamma \cdot \delta; c$ must be in Predict($\bigcup_{l<j} t_{l,j}$).

□

With these lemmas, it is easy to show that our algorithm produces a least-cost repair of the input.

Theorem 2.1. A least-cost repair of w_n is represented by the least-cost dotted production $S \rightarrow S\$ \cdot; c$ in $t_{0,n}$ (where $S \rightarrow S\$$ is the augmented goal production, and the end-marker, \$, is a_n).

Proof. From lemmas 2.4 and 2.5 we know that any $S \rightarrow S\$ \cdot; c$ in $t_{0,n}$ matches w_n and follows λ, and that the least-cost version must be in $t_{0,n}$. Therefore there must be a repair, M, such that $S \Rightarrow^* M(w_n)$ and $C(M(w_n)) = c$. There can be no dotted productions in $t_{0,n}$ with lower cost, since \$ appears only in one production.

□

As mentioned above, in order to keep the sizes of the matrix cells finite, we must discard dotted productions with higher costs. Since our proof of the algorithm established, and required, only that the lowest-cost versions of the dotted productions need be present in the matrix, it is clear that the higher cost versions need not be retained.

2.4. An algorithm for regionally least-cost repair

We have presented an algorithm that finds a least-cost repair of an entire program. We do not propose that it be used as such. Instead, we intend that the repair algorithm be called only when needed, and then only to repair a reasonably sized region of the program. A linear-time parser, such as LL(1) or LR(1), will be

used for the major portion of the program (for the entire program if there are no errors).

Instead of producing a sentence of the language, we want our algorithm to produce a string which can follow the input already accepted by the parser; therefore we will use a grammar that describes the legal continuations. Such a grammar can be derived from the state of the parser. If an LL(1) parser is used, this task is easy: the parse stack describes the expected suffix, and we replace the starting production with the production $S \rightarrow stack$. For an LR(1) (or SLR or LALR) parser, we can use the a modification of technique described in [11] to derive a regular expression that describes the legal suffixes. From this regular expression we can easily derive an equivalent context free grammar, which will be added to the original grammar for the purpose of repair.

The procedure to find a regionally least-cost repair, then, is:

Use the parsing stack to derive a grammar
    that describes the legal continuations.
Define a region, beginning at the current input symbol.
Run the algorithm of the previous section,
    using the derived grammar, on that region.

This procedure computes a repair to the region, following the string already accepted by the parser, since $S \rightarrow \alpha \cdot \beta ; c$ in $t_{0,n}$ represents a modification of the region into a prefix of the language generated by the modified grammar, and since that grammar describes the continuations of the input already accepted. Unfortunately, however, $S \rightarrow \alpha \cdot \beta ; c$ does not necessarily represent a least-cost repair. The problem is that the repair is not necessarily a proper repair, and may contain trailing insertions. Thus, the cost of the repair is inflated by the extra insertions and, more importantly, a lower cost proper repair may be overlooked because the trailing insertions it implies have a higher cost. For example, assume that the legal continuations are described by the following grammar:

$S \rightarrow A$
A → a b c c c
  | d d b f

Assume further that all insertion costs are 1 and that we wish to repair the one-symbol region 'b'. Obviously, the least-cost repair is to insert 'a'; however, the lowest-cost repair, M, such that $A \Rightarrow^* M(b)$, is to insert 'dd' before 'b' and insert 'f' after. It is this repair that will be represented by $S \rightarrow A \cdot ; 3$ in $t_{0,1}$. So even if we strip off trailing insertions to get a proper repair, we are not assured of getting a least-cost repair.

The inclusion of trailing insertions is necessary in the intermediate columns of the matrix, since those insertions must be made in order to accept the next input symbol. It is only when matching the last symbol of the region that the trailing insertions are undesirable. Therefore, we must do things slightly differently when building the last column of the matrix. When j=n, we use the following definitions, instead of those given previously:

$$\text{For } X \in V, a \in V_t, \text{Prefix}(X,a) = \min_{x \in V_t^*,\, b \in V_t} \left[\{\infty\} \cup \{C(x)+RC(a,b) \mid X \Rightarrow^* xby \} \right]$$

$$\text{For } A,B \in V_n, \text{Prefix}(A,B) = \min_{x \in V_t^*} \left[\{\infty\} \cup \{C(x) \mid A \Rightarrow^* xBy \} \right]$$

For Q a set of dotted productions and R a set of symbols,

$$Q \times R = \{ A \rightarrow \alpha B \beta \cdot \gamma ; c \mid A \rightarrow \alpha \cdot B \beta \gamma ; c \in Q, B \in R, c = c \}$$
$$\cup \{ A \rightarrow \alpha \cdot \beta ; c \mid A \rightarrow \alpha \cdot \beta ; c \in Q, c = c + DC(B), B \in R, B \in V_t \}$$

$$Q^* R = \{ A \rightarrow \alpha B \beta \cdot \gamma ; c \mid A \rightarrow \alpha \cdot B \beta \gamma ; c \in Q,$$
$$\text{Prefix}(B,D) \neq \infty, D \in R, c = c + \text{Prefix}(B,D) \}$$
$$\cup \{ A \rightarrow \alpha \cdot \beta ; c \mid A \rightarrow \alpha \cdot \beta ; c \in Q, c = c + DC(B), B \in R, B \in V_t \}$$

For Q and R sets of dotted productions,

$Q \times R = \{A \rightarrow \alpha B \beta \cdot \gamma ; c \mid A \rightarrow \alpha \cdot B \beta \gamma ; c_1 \in Q, B \rightarrow \delta \cdot ; c_2 \in R, c = c_1 + c_2 \}$.

$Q^* R = \{A \rightarrow \alpha B \beta \cdot \gamma ; c \mid A \rightarrow \alpha \cdot B \beta \gamma ; c_1 \in Q,$

$\text{Prefix}(B, D) \neq \infty, D \rightarrow \delta \cdot ; c_2 \in R, c = c_1 + c_2 + \text{Prefix}(B, D) \}$.

When these alternate definitions are used, all of the left operands of × and * (dotted productions of the form $A \rightarrow \alpha \cdot \beta ; c_1$) will come from columns to the left of column j, so their costs will include the necessary trailing insertions, whereas the right operands ($B \rightarrow \sigma \cdot ; c_2$) will all come from column j, and so will not include any trailing insertions. Predict is not changed, since it only involves leading insertions.

Although the procedure above implies that the region is chosen before the repair algorithm is run, it need not be. The two steps may be performed simultaneously. After each iteration of the main loop (for j := 1 to n) in the repair algorithm, the region has been extended by one input symbol, and the matrix contains the least-cost repair to that region. Instead of a for-loop, we can test whether the current region is sufficient, and then either quit or increment j and repeat. Methods for determining the region size are discussed in the next chapter.

This algorithm performs a mostly breadth-first search for the least-cost repair. Some pruning is done as the higher-cost dotted productions are thrown out, but the algorithm still considers a large number of repairs at each input symbol. In practice, simple errors (with simple repairs) will be more common. We can save time overall with a depth-first approach, trying the simple repairs first, and only considering more complex repairs if necessary.

A fast and simple depth-first search is to compute the locally least-cost repair, and 'validate' it, by tentatively parsing ahead using a linear-time parser, as in the repair algorithms of [6, 16, 29]. If the parser reaches the end of the region without detecting additional errors, then the locally least-cost repair is the

regionally least-cost repair and we are done. If another error is detected, then we must fall back to a more powerful algorithm, either the full repair algorithm above, or a depth-first version of it.

We can prune the search done by the repair algorithm by placing a threshold on the cost of the repair, and therefore on the dotted productions. We define new functions, *', ×, and Predict', based on the old functions and the threshold. For example,

$$Q^* R = \{A \rightarrow \alpha \cdot \beta ; c \mid A \rightarrow \alpha \cdot \beta ; c \in Q^* R \text{ and } c \leq \text{Threshold}\}$$

The depth-first repair algorithm is:

```
SetInitialThreshold
repeat
      run the repair algorithm, with modified operators,
        until end of region is reached
        or no repair to input so far is within Threshold
      if no repair within Threshold then
        increase Threshold
        reset input to beginning of region
        endif
until end of region is reached.
```

It is easy to see that this algorithm eventually yields the regionally least-cost repair. At all times, the matrix is a subset of the matrix formed by the breadth-first algorithm; the depth-first matrix contains all those dotted productions with cost less than or equal to the threshold. Since the threshold keeps increasing, it must eventually reach the cost of the least-cost repair, at which point the algorithm will terminate with the least-cost repair contained in the matrix.

The advantage of the depth-first search is that there will be fewer items in each cell of the matrix, so each * and × product will be faster to compute. Offsetting this advantage is the possibility of running through the basic algorithm

several times before the threshold is high enough to include the repair.

In the worst case, the modified algorithm may get almost to the end of the region before being blocked by the threshold. The number of times the process must be repeated before the repair is found depends on how the increment for the threshold is chosen. A fixed increment may be used, in which case the number of iterations depends on the cost of the repair. We may also make the increment dependent upon the state of the algorithm. For example, at the point the repair algorithm is blocked, we might temporarily lift the threshold requirement, and find out what repairs are possible, *at that point*, to extend the region one symbol. We will then know one repair to the region, and can set the threshold to that cost. We could also simply raise the threshold by the cost of deleting the next symbol. In either case, we are guaranteed that each iteration will extend the repairable region by one symbol; the number of iterations will be bounded by the lesser of the region size and the cost of the least-cost repair.

The initial value of the threshold presents a similar set of choices. Besides a fixed value, we might choose the cost of the locally least-cost repair. If we wish first to try the validation scheme above, we can use it to help choose the initial threshold. Instead of giving up validation when a second error is detected, we can continue making local repairs until the end of the region is reached. We can then use the total cost of these local repairs as the threshold, and guarantee that only one iteration will be necessary, since a repair to the region at this cost is already known. The cost of deleting the entire region is another possible threshold value. It carries the same guarantee as the sum of local repairs, and may be easier to compute; but it is likely to be higher, and so will not limit the search as much.

Chapter 3 - Determining the extent of the repaired region

3.1. Goals

In order to evaluate criteria for determining region size, we must have a clear idea of our objectives in performing regionally least-cost repair.

Our primary goal is to improve on the locally least-cost repair model. "Better repair," however, is a broad goal, and any regional repair is likely to be as good as or better than a local repair. A fixed region, even a region of only two symbols, may give good results. However, there will be cases in which the fixed size is too small, providing no improvement over the local repair, and cases in which it is too big, resulting in wasted computation. One may also define heuristics that provide a dynamic region size. Practical tests will show which fixed size or heuristic provides the best combination of quality and speed. We would like, however, something with a more theoretical basis.

A more ambitious and narrowly defined goal is to find the "best" repair. To accomplish this, we must continue expanding the region as long as there is a possibility of gathering more information relevant to the repair. For efficiency's sake, we should also quit expanding the region as soon as that possibility disappears. If we choose the final region correctly, a series of regional repairs will combine to give a globally least-cost repair. Our goal, then, is to find a way of deciding whether relevant information can be gathered by extending the region.

3.2. Equivalent Repairs

A criterion that meets our requirements is that suggested by Levy [25]. His idea was to expand the region until all of the possible repairs are "equivalent," in that they accept the same suffixes.

Definition. Two repairs, M and N, of $x \in V_t^*$ following $y \in Pr(G)$ are *equivalent* if for all $z \in V_t^*$, $yM(x)z \in L(G)$ <--> $yN(x)z \in L(G)$.

It is easy to see that the equivalence criterion does give us the best repairs; if all repairs are equivalent, examining additional input cannot produce additional information. Unfortunately, this seemingly perfect test is not always obtainable in general. The problem of testing two repairs for equivalence is reducible to testing two context-free grammars for language equivalence, which is undecidable.

But there is still hope. Although we cannot give necessary and sufficient conditions for equivalence of repairs in general, we can give sufficient conditions that may work in practice. Levy points out a sufficient condition: that all repairs leave the parser in the same state. This test is easy enough to make, given the full matrix of the repair algorithm. If we use the depth-first modification of the algorithm, however, we will not have access to all of the repairs. If we are parsing ahead to validate the local repair, we will not have the matrix at all. We would like to find a condition for equivalence that is easy to test without knowing all the repairs.

Pai and Kieburtz [29] defined "fiducial symbols" to control the extent of their "context recovery." They could not give an exact test for strong fiducial symbols (that property is also undecidable in general), but they did give a testable property, strong phrase-level uniqueness (SPLU), which is sufficient to prove fiduciality. Tai [36] improved on the SPLU property with his predictors of the grammar, and showed that any symbol that has SPLU is a suffix predictor. In order to define predictors, Tai first defines 'canonical prefix' and 'canonical suffix', based on rightmost and leftmost canonical derivations. A terminal symbol is a prefix (suffix) predictor of the grammar if it has a unique canonical prefix (suffix). Thus, if a is a prefix (suffix) predictor, and α is its canonical prefix (suffix), then

in every rightmost (leftmost) derivation of a string containing a there must be a point at which the sentential form is $\alpha a \beta$ ($\beta a \alpha$) for some β.

Tai's predictors depend on the particular grammar used. We can take the idea one step further, and define predictors of the language.

Definition. A symbol, $a \in V_t$, is a suffix predictor of a language, L, if $x_1 a y_1 \in L$ and $x_2 a y_2 \in L$ implies $x_1 a y_2 \in L$ and $x_2 a y_1 \in L$.

Once again the general problem of finding suffix predictors of a language is undecidable, but we can give sufficient conditions. Tai presented algorithms for finding prefix and suffix predictors of a grammar; these symbols are also suffix predictors of the language.

Theorem 3.1. If a is a prefix predictor in G, then a is a suffix predictor in L(G).

Proof. Let α be the canonical prefix for a, and let uav and $xay \in L(G)$. Then $S \Rightarrow^* \alpha ay$, $S \Rightarrow^* \alpha av$, $\alpha \Rightarrow^* u$, and $\alpha \Rightarrow^* y$. Therefore $S \Rightarrow^* \alpha ay \Rightarrow^* uay$ and $S \Rightarrow^* \alpha av \Rightarrow^* xav$.

□

Theorem 3.2. If a is a suffix predictor in G, then a is a suffix predictor in L(G).

Proof. Analogous to the proof of Theorem 3.1.

□

The advantage of using predictors of the language is that both suffix and prefix predictors of the grammar will serve. Thus we have a potentially larger set of symbols. Having found a set of suffix predictors, we can use them to control the extent of the region of repair.

Theorem 3.3. If a is a suffix predictor of L(G), then all repairs of $ya \in V_t$ following

$x \in Pr(G)$ that do not delete a are equivalent.

Proof. Let M and N be two such repairs, and let $xM(ya)z_1 = xuaz_1 \in L(G)$ and $xN(ya)z_2 = xvaz_2 \in L(G)$. Then since a is a suffix predictor of L(G), $xuaz_2 \in L(G)$ and $xvaz_1 \in L(G)$.

□

We now have a simple method of controlling the region size: Precompute a set of suffix predictors of the language, and expand the region until a symbol in the set is encountered.

The definition of suffix-predictors of a language is more liberal than the definition of SPLU or of predictors of a grammar. The set of suffix-predictors of a language contains all of the prefix- and suffix predictors of the grammar. Nevertheless, it turns out that the set of suffix predictors for a practical language is likely to be small or empty. Tai showed that a symbol that occurs within a self-embedding construct cannot be a predictor of the grammar. The same is not necessarily true of predictors of the language (it is easy to find ambiguous grammars in which this is not true), but it certainly applies to the common programming languages. In Pascal, the only suffix predictor is the keyword **program** (and the end-marker).

Not only are our marker symbols elusive, but the underlying goal, equivalence of all repairs, is also unattainable in practice. In Pascal, for instance, one possible action (as a portion of a repair) is to insert a string of **begins** somewhere in the region. Such a repair will require a suffix that contains matching **ends**, whereas other repairs will not allow the **ends**. The repairs will not be equivalent until the end-marker is read, so we must examine (nearly) the entire input in order to find the best repair. This is a distressing situation; we have already stated that global

repair algorithms are undesirable.

3.3. Revised Goals

Our problems in finding a (small) region that yields the "best" repairs stem from certain recursive constructs in typical programming languages. We must find a way to avoid the effects of these constructs.

Another problem hinders the search for the best repair: the discrepancy between the point of error and the point of error detection. Since the repair algorithm is not called until an error is detected, we must have a way to back up so that the point of error is included in the region, or else accept a sub size 10 -optimal repair of such delayed errors.

The problematic repairs to recursive structures really fall into this same category. Although inserting a string of **begins** is always a possible repair, it can never be least-cost unless a matching string of **ends** is also in the region. We can consider insertion of such a string to be an attempt to repair an error that has not yet been detected. Our revised goal then becomes to find the 'best' repairs to all errors that occur and are detected within the region. If we achieve this goal, then our regional repairs will combine to give a globally least-cost repair if all points of error and error detection coincide, or if we also have a backup algorithm to insure that the point of error is within the region.

3.4. Equivalence of a Reduced Set of Repairs

The criterion that repairs be equivalent is still applicable, but now we test equivalence only on those repairs that do not anticipate undetected errors. Therefore we need some way of separating the repairs.

Let REPAIRS(x,y) be the set of all repairs to y following x. Then we define a reduced set of repairs, MINIMAL(x,y), as follows.

$S(x,y) = \{ M \mid M \in REPAIRS(x,y), \exists N \in REPAIRS(x,y)$ such that
N(y) can be modified into M(y) using only
insertions, and $C(N(y)) \leq C(M(y)) \}$
$MINIMAL(x,y) = REPAIRS(x,y) - S(x,y)$

S contains all those repairs that have "superfluous" symbols. Since costs are non-negative, repairs without superfluous insertions must have costs equal to or less than the corresponding repairs in S. Thus we can never exclude a least-cost repair. The reason for requiring that the shorter repair also have lower cost is to prevent a repair that deletes a symbol from excluding a repair that does not. In particular, if $M(ya)=za$ is a repair of ya following x then $N(ya)=z$ is also a repair. We do not want N to exclude M from MINIMAL. A deletion such as this is the only case in which a repair with a longer resulting string can have lower cost.

We are essentially saying that, by definition, the repairs in S are repairs that anticipate errors not yet detected. (Included in S are also those repairs that insert an optional construct, such as 'id[constant]' instead of simply 'id.' These repairs could be said to repair errors that are undetectable.)

If the detected error really is an error in a recursive structure, the repair will not be excluded. For example, for the Pascal fragment

... **procedure** foo; **if** i < j ...

the repair that inserts **begin** before the **if** is a minimal repair, although the repair that inserts two **begins** is not. An inversion of the actual error may not be a minimal repair. Again in Pascal:

```
if i<0
  goto 1 end;
```

The error is omission of **then begin**, but simply inserting **then** is a possible repair. In such a case we say that there are in fact two errors, one of which is not detectable until later. Insertion of the **begin** remains 'superfluous' until the **end** enters the region; at that point the second error has been detected.

MINIMAL, then, is our set of repairs that do not anticipate undetected errors, and we need to test whether the repairs in that set are equivalent. Symbols that possess Pai and Kieburtz's property "weak phrase-level uniqueness" will almost suffice as markers.

Definition. If A is a non-terminal of G, then G(A) is the (reduced) grammar obtained from G after A is made into a terminal symbol by removing all productions with A as the left-hand side.

Definition. If a is a terminal of G, then a has weak phrase-level uniqueness (WPLU) in G if

a) a is the goal symbol. (G is really $G(a)$)

or

b) there is only one production $A \rightarrow \alpha a \beta$ that has a in the right-hand side, and
   i) a occurs only once in $\alpha a \beta$
   ii) A has WPLU in G(A)

This definition requires that symbols with WPLU have a unique origin, but allows recursion. The problem is that there may be two (or more) distinct recursive derivations, $A \Rightarrow^{*} \alpha A \beta$, that accept different suffixes. For example, in the following grammar:

```
S → A A → a b A a
  | c b A c
  | d
```

the symbol 'd' has WPLU. However, if we wish to repair the two symbol region "b d", we find two repairs in the reduced set: insert 'a' or insert 'c'. The suffixes allowed by the two repairs are clearly not the same. We need to strengthen the definition somewhat, to control the kinds of recursion that can occur. We cannot forbid multiple recursion paths, since such constructs are common in programming languages. Instead, we must insure that any two repairs involving such recursion either accept the same suffixes or else cannot both occur in the reduced set of

repairs.

Definition. A derivation, $A \Rightarrow^* \alpha X \beta$, is a *middle-most* derivation of X from A if in the corresponding derivation tree the only interior nodes (expanded non-terminals) are ancestors of the leaf node labelled X.

The middle-most derivation contains only those steps necessary to reach a certain symbol. Two different derivations are therefore easier to compare. In particular, it is much easier to compare the suffixes allowed by two middle-most derivations of some symbol, since the trailing parts of the sentential forms will be left as general as possible.

Definition. If $\alpha \in V^*$, then First $(\alpha) = \{a \in V_t \mid \alpha \Rightarrow^* a\beta\}$

Definition. A middle-most derivation, $A \Rightarrow^+ \alpha A \beta$, is a *minimal recursive derivation* of A if in the corresponding derivation tree no two ancestors of the leaf node A are labelled by the same symbol.

In defining a modification of WPLU, we must restrict the ways in which recursive derivations can occur. In order to talk about recursive derivations, we confine our attentions to minimal recursive derivations.

Definition. A symbol, a, has Moderate Phrase-Level Uniqueness (MPLU) in G if

a) a has WPLU in G

and

b) if $A \rightarrow ...a...$, then for any minimal recursive derivation, $A \rightarrow \alpha B \beta \Rightarrow^* \gamma A \delta$,

i) if $A \rightarrow \alpha B \beta \Rightarrow^* \gamma A \delta$ is minimal, then $\gamma = \gamma$ and $\delta = \delta$

ii) if $C \rightarrow \alpha D \beta$, is a production in the grammar, distinct from $A \rightarrow \alpha B \beta$, and D $\Rightarrow^*$ ...a..., then First$(\alpha) \cap$ First$(\alpha) = \emptyset$

MPLU adds to WPLU the restriction that any two distinct derivations in which the parents of a may participate must be distinguishable by the first symbol. That way, we can guarantee that two different repairs cannot induce different recursive derivations, and therefore that symbols with MPLU are sufficient to guarantee suffix equivalence among repairs in MINIMAL.

Lemma 3.1. Consider two different middle-most derivation trees corresponding to $S \Rightarrow^* \alpha a \beta$ and $S \Rightarrow^* \gamma a \delta$, where a has MPLU. Let A label an ancestor of a such that the steps from A to ...a... are identical in both derivations, but the step that generates the node labelled A is different between the two derivations. Then in at least one of the derivations there is another node labelled A as an ancestor earlier in the derivation.

Proof. Let $B \rightarrow \alpha A \beta$ and $C \rightarrow \gamma A \delta$ be the steps that generate A in the two derivations. Since a has MPLU, A has WPLU in G(A), and therefore there is only one production, $D \rightarrow \sigma A \tau$, in G such that $S \Rightarrow^* ...D...$ without going through A. (If S can derive D without going through A, then $D \rightarrow \sigma A \tau$ is in G(A). Since A has WPLU in G(A), there is only one such production.) Since $B \rightarrow \alpha A \beta$ and $C \rightarrow \gamma A \delta$ are different, at most one of them can be reachable from S without going through A. Therefore, in that derivation, there must be another node labelled A higher in the tree.

□

Theorem 3.4. If a has MPLU, then all repairs in MINIMAL(u, va) that do not delete a are equivalent.

Proof. Let M and N be repairs in MINIMAL(u, va). For convenience, define two (improper) repairs, M' and N', such that M'(v)a = M(va) and N'(v)a = N(va). (Since the repairs do not delete a, it must be the last symbol in both repaired strings. The repairs M' and N' allow us to show a explicitly in the derivation.) Choose x and

y such that $uM'(v)ax$ and $uN'(v)ay$ are sentences in L(G), and consider derivation trees corresponding to $S \Rightarrow^* uM(v)ax$ and $S \Rightarrow^* uN(v)ay$. We will examine the ancestors of a in both derivations, showing that the middle-most derivations from S to a must be the same in both cases, and therefore the suffixes accepted must also be the same.

As in Lemma 3.1, we work backwards in the middle-most derivation trees of a until we reach a step that is not the same in both derivations. Let the last common ancestor be labelled by A. By Lemma 3.1, one of the derivations has another, higher node labelled A; call that derivation "derivation 1." Now consider the next lowest node that is also labelled by A, and the first symbol in the repaired input string that derives from this node, call it b. If the symbol is not in the correct prefix, w, but in the repaired portion of the input, then the repair is not minimal, since we can remove all the symbols derived within the loop between the two A nodes (and remove the loop from the derivation tree), and still have a repair. Otherwise, the symbol must be part of the correct prefix, w. Since a has MPLU, A is an ancestor of a, and b is the first symbol in a recursive derivation of A, we know that the lowest node in derivation 2 that is an ancestor of both a and b must be labelled A. Furthermore, we know that in both trees, the minimal derivation from A to A must have the same prefix and suffix. The only way in which the suffixes could differ is that one derivation contain additional loops between the two nodes labelled A. Again we can look at the first terminal symbol derived from the loop, and show that the loop is either unnecessary or also present in the other derivation. Therefore the suffix accepted by the derivation from A to a must be the same in both cases.

The portion of the derivation from S to our node labeled A covers the same input in both trees. Therefore it doesn't really matter how those steps of the

derivations proceed; the possible steps in derivation 1 must also be possible in derivation 2, so the suffixes accepted must be same in both trees.

□

The additional restrictions of MPLU are not a hardship in practice; all of the Pascal symbols mentioned by Pai and Kieburtz as having WPLU also have MPLU. In Pascal, the major recursive constructs, the compound statements, all begin with distinctive keywords: **with, while, repeat, for, begin, case.** For our purposes, a symbol need only have MPLU in *some* grammar for the language, not necessarily in the grammar actually used. For example, a typical grammar for Pascal might describe statements as follows:

```
<STMT> ::= for <ID> := <EXPR> <TO> <EXPR> do <STMT>
         | while <EXPR> do <STMT>
         | with <RECORD LIST> do <STMT>
         ...
```

Obviously, the keyword do does not have WPLU in this grammar, since it appears in three different productions. It does have WPLU (and can have MPLU), however, in this grammar:

```
<STMT> ::= <STRUCT STMT HEADER> do <STMT>
<STRUCT STMT HEADER> ::= for <ID> := <EXPR> <TO> <EXPR>
                       | while <EXPR>
                       | with <RECORD LIST>
                       ...
```

When fiducial symbols are chosen, more than one grammar for the same language may be examined. Once the set is fixed though, the working grammar may be changed without endangering the repair algorithm (so long as the language is not changed).

Chapter 4 - Implementation results

4.1. Implementation

The original GHR algorithm is cubic in the length of the input and linear in the size of the grammar [17]. (Ruzzo [34] discusses sub-cubic versions, but admitted that they are not practical.) Since our algorithm is more complicated, we cannot hope to do better than this, and we must be careful not to do worse. It is clear that algorithm 2.0 must require time at least $O(n^3)$, since there are two explicit loops on lines 3 and 6, and an implicit loop on line 8. The complexity of the algorithm also depends on the time needed to compute the ×, *, and PREDICT functions. As we will show, the complexity of these functions does not depend on the number of input symbols, but does depend on the size of the grammar.

In order to represent the cells of the parse matrix, we first assign an index to each dotted production (without costs), $A \rightarrow \alpha \cdot \beta$, as suggested in [17]. Indices are assigned so that Index($A \rightarrow \alpha X \cdot \beta$) = Index($A \rightarrow \alpha \cdot X \beta$)+1; auxiliary data structures insure that, given an index, the symbol to the right of the dot and the symbol on the left-hand side can be quickly determined. An item in a cell of the matrix, $A \rightarrow \alpha \cdot \beta ; c$, can then be considered as an ordered pair, (Index($A \rightarrow \alpha \cdot \beta$), c). In our implementation, each cell is represented by an array, indexed by the dotted production and containing the corresponding cost. Thus, Cell[Index($A \rightarrow \alpha \cdot \beta$)] = c. If a dotted production is not present in the cell, it is given a cost of "infinity." A similar array contains the costs of completed left-hand sides. That is, if $A \rightarrow \alpha \cdot ; c$ is in the cell, then Completed[A] = c. As mentioned in Chapter 2, only the lowest cost example of a dotted production is needed. Similarly, if there are two different dotted productions with A as the left-hand side, $A \rightarrow \alpha \cdot ; c_1$ and $A \rightarrow \beta \cdot ; c_2$, the higher cost dotted production will never be used in a least-cost repair.

Therefore the arrays are sufficient to represent the cells of the parse matrix. A simple check whenever a new dotted production is added to the cell will insure that the cost stored is minimal.

Using this representation, it is easy to implement the pasting operators. Let us look first at the × operator. To paste a single dotted production, $A \rightarrow \alpha \cdot X \beta ; c_1$, with index i, to another dotted production, $C \rightarrow \sigma \cdot ; c_2$, with index j, we simply check that X=C (if not, the result is empty), then add the costs and store the sum in the result array, at location i+1. That is the first phase of the operation, which we may call the direct match. The second phase involves moving the dot across the remaining symbols in the right-hand side, yielding dotted productions $A \rightarrow \alpha X \beta_1 \cdot \beta_2 ; c_1 + c_2 + C(\beta_1)$. Since the indices of the dotted productions are assigned in order, this dot movement is done by a simple iterative loop, adding the cost of the next symbol to the running total and storing it in the next location. When the dot reaches the end of the right-hand side, we add the final cost of A to the 'Completed' array.

To paste a single dotted production, $A \rightarrow \alpha \cdot X \beta ; c$, to a set of dotted productions, we simply extract the cost of X from the 'Completed' array, and proceed as above. To paste two sets of dotted productions together, we use a simple iterative loop to examine all the elements in the left operand, and paste single dotted productions to the set in the right operand. Now we are merging the results of several individual pastings, so we must check whether the new result is lower cost before writing it into the array.

In order to paste dotted productions to input symbols, we place a cost of zero in the 'Completed' array for the input symbol. Whenever we paste a dotted production to an input symbol, we can include deletions by adding the deletion cost of the symbol and not changing the index.

Once we begin merging results of individual pasting operations, we find that the efforts expended in dot movement phase may be duplicated. When $A \rightarrow \alpha \cdot \beta ; c_1$ is pasted to $C \rightarrow \sigma \cdot ; c_2$, $A \rightarrow \alpha \beta_1 \cdot \beta_2 ; c_3$ will be added as part of the dot movement phase; if $A \rightarrow \alpha \beta_1 \cdot \beta_2$ is added again at lower cost, then all the dot movement within β_2 will be repeated. In the worst case, this repeated movement can result in work proportional to the square of the size of the grammar. However, it is not necessary to perform dot movement immediately after each direct match. Instead, we can defer dot movement until all the direct matches have been performed, so that we only perform dot movement for the least-cost direct matches. All dot movement can then be done in a single pass through the cell.

The * product is similar to the × product, but has the additional problem of indirect derivations. To paste a single dotted production, $A \rightarrow \alpha \cdot X \beta ; c$, to a set of dotted productions, we must look at all symbols, B, that are derivable from X, and check whether B is completed in the set of dotted productions. The operation then proceeds as in the previous case, but of course the cost includes Derive(X, B). We can speed up the * product by keeping a list for each symbol X of all the symbols it can derive (along with the cost of doing so).

Using this representation, then, we can compute a ×-product in time $O(|G|)$, and a *-product in time $O(|V| \cdot |G|)$. The entire algorithm is therefore cubic in the number of input symbols and quadratic in the size of the grammar. In space requirements, the algorithm is quadratic in the number of input symbols and linear in the size of the grammar. Graham, Harrison and Ruzzo [17] suggest several alternative implementations of their algorithm. Most of them depend on the fact that once a dotted production is added to a set, it need not be added again. In our case, however, we must determine if the dotted production can be added at lower cost, so these methods are not applicable to the repair algorithm.

4.2. Results

We ran separate tests to measure the quality of the regional repairs, the space and time requirements of the repair algorithm, and the typical size of regions. The repair algorithm was implemented as described above, and was capable of repairing regions of fixed size only.

4.2.1. Quality of Repairs

Since the regionally least-cost repair model is an extension of the locally least-cost model, we would expect the quality of such repairs to be higher. However, since the locally least-cost repair algorithm already performs very well, the margin for improvement is not great. In fact, in many cases, the locally least-cost repair is identical to the regionally least-cost repair.

To test our repair algorithm we used a collection of Pascal syntax errors provided by Ripley and Druseikis [32]. They collected programs with syntax errors, classified the errors, discarded duplicates, and condensed the programs into a set of short illustrative examples. The collection represents the different kinds of errors encountered, but not their relative frequencies.

We first ran the locally least-cost repair algorithm [14] on the collection of syntax errors and found that about 28% of the errors were poorly handled. It is on these poorly repaired errors that we can expect improvement from a regionally least-cost repair. We ran the regionally least-cost repair algorithm on the same programs, using the same costs and a fixed, five-symbol region. In 56% of the situations in which the locally least-cost algorithm did poorly, the regionally least-cost algorithm did well. In another 13%, the regionally least-cost algorithm would have done better had the costs been adjusted or the region size increased. The remaining errors, which were handled poorly, fell into three groups. The first group consisted of errors in which the point of error did not coincide with the point of

detection. In order to improve such repairs we must either back up after the error is detected, to insure that the point of error is within the region, or include special error productions in the grammar to handle specific errors. The second group contained "large-scale" errors, such as type and variable declaration sections in the wrong order. These errors cannot be repaired by insertion and deletion of individual symbols. The preferred action is to flag the construct as erroneous, but accept it anyway. Again, individual error productions will handle such errors. The third group of errors consisted of large portions of garbage in the program, usually due to missing or incorrect delimiters for comments and character strings. The best repair is to delete the whole mess. In this instance, the recovery algorithms have an advantage over the repair algorithms. Repair algorithms do not usually delete a long string of symbols; instead they insert more symbols to make a valid construct. The regional repair algorithm makes less of a mess than the local algorithm, but we still cannot consider that there is a significant improvement.

Let us examine two examples in which the regional algorithm would do better than the local. These examples are adapted from the Ripley/Druseikis programs [32].

```
program p (input, output);
var a, b, : real; i, j : integer;
begin end .
```

The error is an extra comma after identifier *b*, but the parser will not detect error until the colon is read. Without backing up the parser, there is no way to remove the comma, so we must either insert another identifier, or delete the colon. The locally least-cost algorithm chose to delete the colon, and a second error was detected when the semicolon was read. This second error was within the region of repair, so the regional algorithm chose to insert an identifier instead.

```
program p (input, output);
funtion f (var x: integer): boolean;
  begin
  end;
begin
end .
```


In this example, an error is announced when 'funtion' is read. Without a spelling corrector, replacing 'funtion' with **function** is not plausible, so we must make do with insertions and deletions. The locally least-cost repair was insertion of **const** (**type** or **var** would do equally well) and the results were disastrous. The regionally least-cost algorithm will see the additional repairs necessary if **const** is inserted, and will instead insert **function** and delete either 'funtion' or 'f'. This example also emphasizes the effect of region size; the five symbol region was not large enough, and in our tests, the regional algorithm chose the same repair as the local algorithm. However, if the region were extended, the repair we suggest would eventually become least-cost.

With costs tuned for regional repairs and with variable region size, the regionally least-cost algorithm should choose acceptable repairs on about 91% of the test cases. Performance could be further improved with the use of specific error productions. These results compare favorably with the locally least-cost algorithm, which had 72% acceptable repairs, and with the Berkeley pascal compiler [22] which performs acceptably on 80% of the cases [16].

4.2.2. Efficiency

The regionally least-cost repair algorithm is fairly large and slow. Our grammar for Pascal has 185 symbols (terminals plus non-terminals), 234 productions, and 640 distinct dotted productions. Since the algorithm considers all combinations of insertions and deletions, almost all of the dotted productions are present in every cell of the parse matrix. On the Ripley/Druseikis programs, cells were either about 95% full or about 44% full, depending on the position of the error in the program. If the error occurred within the declarations portion of the

program, including the procedure and function definitions, then almost all of the productions in the grammar could be reached by way of some repair, and the cells were very full. If the error occurred after the **begin** that marks the start of the main body, then all the productions dealing with declarations were no longer applicable, and the cells were smaller. When the cells were 95% full, the algorithm required approximately 15 seconds of cpu time on a VAX-11/780 to build the table for a five-symbol region; when the cells were 44% full, it took about 9 seconds.

By placing a threshold on the costs, as suggested in Chapter 3, we reduced the densities of the cells significantly. We tuned the threshold values to each error by looking at the repairs chosen when no threshold was imposed. With the thresholds always equal to the cost of regionally least-cost repair, the cell densities ranged from 75% down to less than 2%, with an average of 33%. The time required to build the matrix dropped correspondingly, to as low as one second, with an average of 7.5 seconds. These speeds are probably too slow for use in compilers, but might be acceptable in an application in which the time required would be masked by other effects. For example, in an interactive program, the repair matrix could be built while the input is being typed; the slowness of the algorithm would be matched by the slowness of human fingers.

We expect that we could speed up the algorithm in several ways. Although we were fairly careful in coding the algorithm, there is no doubt that further scrutiny would reveal room for improvement. We also cannot claim that our representation of the parse matrix is the most efficient. Additional speed improvement may come from avoiding use of the full regionally least-cost algorithm when possible, for instance, by validating the locally least-cost repair, and only calling the regional algorithm when the validation fails.

4.2.3. Region Sizes

In order to measure region sizes, we modified the LL(1) locally least-cost repair algorithm to perform a parse-check after choosing a repair. The parse-check parsed forward from the point of repair until it accepted a member of a set of marker symbols, or until a second error was detected. In the case of a second error, the parse check continued scanning (without parsing) until a marker was read. The parse check reported success or failure and the size of the region following the first error.

We ran this modified parser/repair algorithm on the Ripley/Druseikis error programs. However, since these programs are condensed examples of each error, it is not clear whether the region sizes are representative of those to be found in larger programs. In particular, it is impossible to have very large regions in such short programs. Since we did not have a good collection of errors in large programs, we instead measured the distance between markers, with or without errors. We made this region measurement on the Ripley/Druseikis programs, and on a working Pascal program with 1925 lines of code, exclusive of comments (this program was, in fact, the regionally least-cost repair algorithm).

We made our region size measurements using five sets of marker symbols. The first set contained the symbols found by Pai and Kieburtz to have WPLU [29]: **begin, const, do, downto, else, for, goto, if, label, program, repeat, then, to, type, until, while,** and **with.** These symbols also have MPLU. The second set consisted of the elements of the first set, plus the symbol ':='. Although ':=' does not have MPLU (or WPLU), it can only appear in two contexts in Pascal: assignment statements and **for** statements. In order for it to be used in a **for** statement, it must either be preceded by the keyword **for**, which is itself a marker, or the repair algorithm must insert **for**, which is an non-minimal repair by the

definition of Chapter 3. Thus, ':=' seems to be a reliable marker, even though it does not possess MPLU.

To the third set we added still more symbols that do not have WPLU, but which appear in limited contexts: **array, end, function, record, packed, procedure,** and **var.** As we will see, these symbols would be important markers, since no member of the first two sets appears in the declarations section of a program. To the fourth set we added ';'; the fifth set consisted of ';' alone. The results of these region measurements are summarized in Table 1.

As we would expect, the average region size went down as the number of marker symbols increased. The region sizes for the collection of smaller programs were similar to those in the large program. The difference between large and small programs can be seen in the maximum region size; the large program contained a region of 1112 symbols, over twenty times larger than any program in the Ripley/Druseikis collection. This region consisted of all the global type and variable declarations, and the forward procedure declarations. It was nearly ten times the size of the next largest region, which contained 125 symbols. Such a large declaration section is not unusual in Pascal programs; global variables are difficult to avoid, and global type declarations are essential. It is therefore important that the set of marker symbols include symbols found within declarations. When we used marker set 3, the largest region had 125 symbols (the same region mentioned before). This region was further divided by the addition of ';' to the marker set. ';' does not have any of the formal properties discussed in Chapter 4, but it is often used as a marker in recovery algorithms. It certainly does a good job of dividing the program into reasonably sized regions.

The sizes of the regions following errors correspond reasonably with the regions measured independent of errors. Of more interest is the success of the

parse-checks of the locally least-cost repairs. Since the locally least-cost repair algorithm performs very well in most cases, and is much faster than the regionally least-cost algorithm [11,14], it makes sense to use regionally least-cost only when necessary. Tentatively parsing beyond the point of error is one way to determine whether a regional repair is necessary. As stated before, if the parser detects no additional errors before the end of the region is reached, then the locally least-cost repair is the regionally least-cost repair. In our tests, the parse-check succeeded about 50% of the time, the success rate increasing somewhat as the average region size decreased. Since the time to compute a locally least-cost repair and to parse-check is very small compared to the time to compute a regionally least-cost repair, we could effectively double the speed of the algorithm, without affecting the repairs at all.

The 50% success rate of the parse-check is lower than the 72% rate of good repairs mentioned earlier. Therefore the parse-check must be failing on some of the repairs that we have called acceptable. This difference is due to the presence of additional errors in the same region (despite Ripley's and Druseikis's efforts to have only one error per program), and to the nature of the local repair algorithm. Some errors are repaired in two steps, with two separate calls to the repair algorithm. For example, since the algorithm does not perform transpositions directly, it must handle transposed symbols by deleting the first, returning to the parser, and, when called again, reinserting the deleted symbol (the same effect could also be accomplished by inserting, then deleting). The error is repaired properly, but a parse check after the first phase would fail.

		Marker Set 1	2	3	4	5
Large Pascal Program						
Region size	mean	10.34	7.26	5.17	3.30	6.67
	median	3.13	3.00	2.33	1.69	4.37
	maximum	1112	1112	125	49	71
Ripley/Druseikis Programs						
region size	mean	7.08	5.26	4.99	3.48	8.29
	median	5.33	3.11	3.18	1.80	7.04
	maximum	49	49	49	31	33
Parse-check success		48.8%	51.0%	53.0%	61.0%	54.2%
Parse-check region size						
	mean	4.40	3.79	3.38	2.85	4.14
	median	2.25	1.95	1.88	1.73	2.48
	maximum	20	20	14	12	17

Notes:

set 1 = **begin, const, do, downto, else, for, goto, if, label, program, repeat, then, to, type, until, while, with.**

set 2 = set 1 plus ':='

set 3 = set 2 plus **array, end, function, record, packed, procedure, var.**

set 4 = set 3 plus ';'

set 5 = ';'

Parse-check region sizes are distance from the point of error detection to the next marker in the input. Other region sizes are distance between markers.

Table 1

Chapter 5 - Conclusions

5.1. Summary

We have presented a definition of "regionally least-cost repair," and an algorithm to find such repairs. Our algorithm adapts the approach of Aho and Peterson [1] and Lyon [27] to the new context-free parser of Graham, Harrison, and Ruzzo [17]. The algorithm can be used to find a least-cost repair to any region of the input string. Although the idea of regional repair is not new [35, 25], ours is the first implementation.

Our repair algorithm permits replacements, as well as insertions and deletions. We have shown that the use of replacements in the algorithm poses no serious problems. In fact, replacement operations can be added to the locally least-cost repair algorithms [11, 14] with little effort, since almost all of the work is done by the 'Derive' function.

To perform a regionally least-cost repair, we must determine the extent of the region. The region begins at the point of error detection, or, if a backup algorithm is used, somewhere to the left of that point. We have shown that Levy's test for the end of the region, that all repairs be equivalent, is impractical. Instead, we suggest that the region end when all members of a certain class of repairs are equivalent. In this way we exclude repairs that make equivalence impossible by unnecessarily increasing the depth of recursive constructs. Symbols that have the moderate phrase-level uniqueness property can be used as region delimiters; their presence as the last symbol in the region insures that all minimal repairs are equivalent. Pascal has many such symbols, and they divide a program into regions of reasonable size.

5.2. Directions for future research

Our repair algorithm, as it stands, is too slow. Significant improvements in speed must be made before it can be used in production compilers. Although the implementation described in chapter 4 seems reasonable, we cannot claim that it is the most efficient. Graham, Harrison, and Ruzzo presented several different implementations for their algorithm [17], none of which was clearly superior. Although, most of their suggestions are not immediately applicable to the repair algorithm, further research may show how to adapt one of their suggestions, or a new idea, to provide a more efficient implementation.

Even if the implementation is improved, the algorithm will be relatively slow, since it must consider so many repairs. However, in most cases the full algorithm is unnecessary. Currently, the parse-check of a local repair is the only technique we have to determine whether the regional algorithm is really needed. We would like to have a way to tell whether two errors are independent. If they are independent, we can repair them separately, without using the regional repair algorithm.

More work is also needed on the problem of region boundaries. Symbols with MPLU make fairly good region markers: they guarantee that the region is large enough, within the limits discussed in chapter 3, and, in Pascal, they divide the program into reasonably small regions, on average. However, in Pascal they also leave one very large gap -- the global declaration section. From a practical standpoint, we must break the declarations into smaller regions, preferably without affecting the quality of the repairs. From a theoretical standpoint, we would like to find a characterization of region markers that includes symbols like ':=' and **array** in Pascal, without giving up the guarantees provided by symbols with MPLU. A better understanding of the independence of errors may help here as well.

Finally, we point out that use of our repair algorithm is not limited to compilers. The locally least-cost repair algorithm has already been adapted for use in a language-based editor [12]. Our regional algorithm could be similarly adapted, although it would be more difficult. The combination would result in a very powerful program development tool, which would allow the user to create a program by specifying only the important features; the remaining details would be filled in by the repair algorithm.

References

[1] Aho, Alfred V. and Thomas G. Peterson , "A minimum distance error correcting parser for context-free languages," *SIAM Journal of Computing 1*, 4, pp. 305-312 (1972).

[2] Aho, Alfred V. and Jeffrey D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Mass. (1977).

[3] Anderson, S. O. and Roland C. Backhouse, "Locally least-cost error recovery in Earley's algorithm," *ACM Transactions on Programming Languages and Systems* 3, 3, pp. 318-347 (July 1981).

[4] Backhouse, Roland C., *Syntax of Programming Languages, Theory and Practice*, Prentice-Hall (1979).

[5] Begwani, Vimal, "A New Approach for Attribute Evaluation and Error Correction in Compilers," Ph.D. Thesis, University of Wisconsin (August, 1982).

[6] Burke, Michael and Gerald A. Fisher, "A Practical Method for Syntactic Error Diagnosis and Repair," *SIGPLAN Notices 17*, 6, pp. 67-78 (June 1982).

[7] Conway, R. W. and T. R. Wilcox, "Design and implementation of a diagnostic compiler for PL/I," *Communications of the ACM 16*, pp. 169-179 (1973).

[8] Dion, Bernard A., *Locally least-cost error correctors for context-free and context-sensitive parsers*, UMI Research Press, Ann Arbor (1982).

[9] Druseikis, Frederick C. and G. David Ripley, "Extended SLR(k) Parsers for Error Recovery and Repair," *Proceedings of the ACM Annual Conference*, pp. 396-400 (1976).

[10] Feyock, S. and P. Lazarus, "Syntax-directed correction of syntax errors," *Software Practice and Experience* 6, pp. 207-219 (1976).

[11] Fischer, Charles N., Bernard A. Dion, and Jon Mauney, "A Locally Least-Cost LR Error-Corrector," *ACM Transactions on Programming Languages and Systems*, (to appear).

[12] Fischer, Charles N., Greg Johnson, and Jon Mauney, "An Introduction to Editor Allan Poe," Tech. report #451, University of Wisconson-Madison (1981).

[13] Fischer, Charles N. and Jon Mauney, "On the role of error productions in syntactic error correction," *Computer Languages* 5, pp. 131-139 (1981).

[14] Fischer, Charles N., Donn R. Milton, and Jon Mauney, "A locally least-cost LL(1) error corrector," Tech. Report #371, University of Wisconsin (August 1979).

[15] Fischer, Charles N., Donn R. Milton, and Sam B. Quiring, "Efficient LL(1) error correction and recovery using only insertions," *Acta Informatica 13*, 2, pp. 141-154 (1980).

[16] Graham, Susan L., Charles B. Haley, and William N. Joy, "Practical LR error recovery," *Sigplan Notices 14*, 8, pp. 168-175 (1979) Proceedings of the Sigplan Symposium on Compiler Construction.

[17] Graham, Susan L., Michael A. Harrison, and Walter L. Ruzzo, "An Improved Context-Free Recognizer," *ACM Transactions on Programming Languages and Systems* 2, 3, pp. 415-462 (July 1980).

[18] Graham, Susan L. and Steven P. Rhodes, "Practical syntactic error recovery," *Communications of the ACM 18*, pp. 639-650 (1975).

[19] Gries, David, "The use of transition matrices in compiling," *Communications of the ACM 11*, pp. 26-34 (1968).

[20] Irons, E. T., "An error-correcting parse algorithm," *Communications of the ACM* 6, pp. 669-673 (1963).

[21] James, L. R., "A syntax directed error recovery method," Tech. Report CSRG-13, Computer Systems Research Group, University of Toronto (May 1972) M.S. thesis.

[22] Joy, William N., Susan L. Graham, and Charles B. Haley, "Berkeley Pascal user's manual version 1.1," , University of California, Berkeley (1979).

[23] Krawczyk, Tomasz, "Error Correction by Mutational Grammars," *Information Processing Letters 11*, 1, pp. 9-15 (August 1980).

[24] Leinius, R. P., "Error detection and recovery for syntax directed compiler systems," Ph.D. thesis, University of Wisconsin-Madison (1970).

[25] Levy, J. P., "Automatic correction of syntax errors in programming languages," *Acta Informatica 4*, pp. 271-292 (1975).

[26] Lewi, J., K. DeVlaminck, J. Huens, and M. Huybrechts, "The ELL(1) parser generator and the error recovery mechanism," *Acta Informatica 10*, pp. 209-228 (1978).

[27] Lyon, G., "Syntax-directed least-errors analysis for context-free languages: a practical approach," *Communications of the ACM 17*, pp. 3-14 (1974).

[28] Mickunas, M. D. and J. A. Modry, "Automatic error recovery for LR parsers," *Communications of the ACM 21*, pp. 459-465 (1978).

[29] Pai, Ajit B. and Richard B. Kieburtz, "Global Context Recovery: A New Strategy for Parser Recovery From Syntax Errors," *ACM Transactions on Programming Languages and Systems* 2, 1, pp. 18-41 (January 1980).

[30] Pennello, Thomas J. and Frank L. DeRemer, "A forward move algorithm for LR error recovery," *Fifth ACM Symposium on Principles of Programming Languages*, pp. 241-254 (1978).

[31] Poplawski, D. A., "Error recovery for extended LL-Regular parsers," Ph.D. thesis, Purdue University (August 1978).

[32] Ripley, G. David and Frederick C. Druseikis, "A Statistical Analysis of Syntax Errors," *Computer Languages 3*, pp. 227-240 (1978).

[33] Roehrich, Johannes, "Methods for the Automatic Construction of Error Correcting Parsers," *Acta Informatica 13*, 2, pp. 115-139 (1980).

[34] Ruzzo, Walter L., "General Context Free Language Recognition," Ph.D. Thesis, University of California, Berkeley (June 1978).

[35] Tai, Kuo Chung, "Syntactic error correction in programming languages," *IEEE Trans on Software Engineering SE-4*, 5, pp. 414-425 (1978).

[36] Tai, Kuo Chung, "Predictors of Context-Free Grammars," *SIAM Journal on Computing* 9, 3, pp. 653-664 (August 1980).

[37] Watt, David A., "Irons' Error Recovery In (LL and) LR Parsers," Technical Report #8, University of Glasgow (August 1976).