A MODIFICATION TO EARLEY'S ALGORITHM FOR SYNTAX ERROR RECOVERY

by

DALE HAGGLUND
B.Sc., University of Alberta, 1986

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in the Department
of
Computer Science

[redacted]

We accept this dissertation as conforming
to the required standard

[redacted]
Dr. R.N. Horspool

[redacted]
Dr. G.C. Shoja

[redacted]
Dr. W.D. Little

[redacted]
Dr. R. Vahldieck

ABSTRACT.

The problem of syntax error recovery has recently received a great deal of attention. Most methods in common use are largely *ad hoc*. Because of this, they tend to have certain pathological cases which cause cascades of error messages. One method of avoiding these error cascades involves using a suffix parser — one which can parse an arbitrary *suffix* of the specified language. Once an error is detected, the parser is simply restarted to look for subsequent errors. Thus, no spurious error messages are ever generated. Earley's algorithm provides the basis for a general suffix parser.

Using a suffix parser as described above allows recovery from syntax errors with no attemp at repair. The modified Earley parser leaves a large amount of information behind in the states it builds to describe its parsing actions which can be used to guide the search for potential repairs. An algorithm based on this fact is presented, along with a simple implementation.

Examiners:

[redacted]

Dr. R.N. Horspool

[redacted]

Dr. G.C. Shoja

[redacted]

Dr. W.D. Kinlc

[redacted]

Dr. R. Vahldieck

TABLE OF CONTENTS.

LIST OF FIGURES.

ACKNOWLEDGEMENTS.

I would like to thank my supervisor, Dr. Nigel Horspool, for his support of this work, as well as for his amazing patience.

CHAPTER 1

INTRODUCTION.

1.1. Syntax Errors

Consider the following scenario:

> After making the final changes to her program, Polly Programmer saves the file, and begins to compile it. However, instead of the "compilation successful" message she was expecting, she sees dozens of messages like
>
> ```
> "main.c", line 59: syntax error at or near symbol (
> ```

Syntactic errors are those that occur when the rules describing the structure of the language are broken. For example, the Pascal fragment

```
a := b + / c
```

contains a syntax error because it is not legal in Pascal for two arithmetic operators to be adjacent to each other. Syntax errors differ from *lexical* and *semantic* errors. Lexical errors occur when an invalid word, or "token," appears, while semantic errors occur when a lexically and syntactically valid phrase has no meaning under the rules of the language. Programs which check for syntactic correctness of programs are called parsers.

Parsers widely used in current compilers, namely $LL(k)$ and $LR(k)$ parsers, halt as soon as they detect a syntax error, and will proceed no further. This is commonly called the *correct prefix property*. This property is useful because it guarantees that an error will be detected as soon as possible in the input stream. Note, however, that only the *first* error in the input is so detected, since the parser will immediately halt. This is disadvantageous, since, typically, we would like to detect all (or at least most) of the

errors in the source stream at one pass. However, the parser is now in an inconsistent state, since the current input token is illegal given the preceding input. Thus, in a traditional compiler, it is necessary to "adjust" the state of the parser in some way before parsing can resume. The decision of just how to make this adjustment is the central issue in the handling of syntax errors.

Through the use of parser generators such as Yacc [Joh75], the construction of parsers has begun to yield to automation. However, error recovery strategies continue to be mainly *ad hoc* solutions to the problem.

1.2. Structure of a Compiler

A compiler consists of three main phases: *lexical analysis*, *syntactic analysis*, and *semantic analysis*. Conceptually, it is convenient to think of these happening sequentially, one after the other. Very few compilers actually work in this manner, however. Generally, the three phases proceed in parallel, with the syntactic analyzer driving the other two.

Lexical analysis breaks up the characters of the input stream into *tokens*, and returns these tokens to the syntactic analysis phase. For example, a lexical analyzer for Pascal, upon seeing the sequence of characters *b*, *e*, *g*, *i*, and *n*, would return a token for the reserved word `begin` to the parser[1]. The lexical analysis phase will usually strip comments from the input, and also remove strings of blanks and newlines. Thus, the parser itself sees only a sequence of tokens, with none of the the textual structure of the original input. The process of *tokenizing* the input has the added advantage that the amount of data handled by the syntactic analyzer is substantially reduced, since long identifiers like `supercalifragilistic` become single tokens.

The syntactic analysis phase takes the stream of tokens produced by the lexical analyzer and ensures that the grammatical rules of the language in question have been

---

[1] It should be noted that a `begin` token would not be recognized in the strings "xbegin," and "begin1," for example.

obeyed. It is with this phase of analysis that the body of this thesis is concerned. The syntactic analyzer reads the stream of tokens, and attempts to build a *parse tree* for the input stream. This parse tree will reflect the structure of the program as defined by the rules of the language. The syntactic analyzer does not, however, attempt to enforce either the static or dynamic semantics of the language.

The final phase of a compiler is that of semantic analysis. In this phase, the parse tree produced by the syntactic analyzer is checked against the semantic rules of the language. The Pascal expression `b + i` is syntactically correct, but may fail to be correct semantically, if, for example, `b` were a boolean variable and `i` an integer variable.

1.3. Terminology

A *grammar* G is a four-tuple (N,T,P,Σ). N is a set of *nonterminal* symbols and T a set of *terminal* symbols. Conventionally, members of N are represented by uppercase letters, while members of T will be represented by lowercase letters. Lower case Greek letters will represent strings of terminals and nonterminals. P is a set of *productions* of the form $X \rightarrow \omega$, where, as mentioned above, ω is a (possibly empty) string of symbols from $N \cup T$. Σ is a special member of N, called the *start symbol*, or the *goal* of the grammar[2]. The following grammar describes all arithmetic expressions involving addition and multiplication with the identifier i as an operand.

$$\begin{array}{ll} \Sigma \rightarrow \vdash E \dashv & T \rightarrow T * F \\ E \rightarrow E + T & T \rightarrow F \\ E \rightarrow T & F \rightarrow i \end{array}$$

Figure 1.1. Grammar for Arithmetic Expressions

We adopt the convention that the goal production always has the form $\Sigma \rightarrow \vdash A \dashv$, where $\vdash$ and $\dashv$ are terminal symbols appearing nowhere else in the grammar, and A is

[2] A more detailed discussion of context-free grammars is given in [ASU86]

some other nonterminal. If a grammar G is not in this form, it can be trivially modified to be so by introducing new nonterminals $\vdash$ and $\dashv$, and adding a rule of the form $\Sigma' \rightarrow \vdash \Sigma \dashv$, where Σ is the old goal symbol, and Σ' is the new goal symbol.

A string $\omega \in (N \cup T)^*$ is said to be a *sentential form.* If such a string is of the form $\omega = \gamma A \phi$, where $A \in N$, and $A \rightarrow \lambda$ is in P, then the sentential form $\omega' = \gamma\lambda\phi$ is said to be *derived* from ω, which we denote by $\gamma A \phi \Rightarrow \gamma\lambda\phi$. We also let $\Rightarrow^*$ denote the reflexive, transitive closure of $\Rightarrow$. $L(G)$, the *language* generated by G is the set of strings of terminal symbols which can be obtained from Σ by some finite sequence of derivations.

1.4. LR Parsers

LR parsers fall into the category of shift-reduce, or *bottom up*, parsers. This means that they build the parse tree from the terminal symbols up, rather than from the goal symbol down, as *LL* parsers do. This section gives a simple explanation of how an *LR* parser works. A more detailed description of *LR* parsers can be found in [AhJ74]

Consider the grammar G below:

$$\Sigma \rightarrow \vdash LIST \dashv$$
$$LIST \rightarrow LIST\ ,\ ELEMENT$$
$$LIST \rightarrow ELEMENT$$
$$ELEMENT \rightarrow a$$
$$ELEMENT \rightarrow b$$

Strings described by this grammar include $\vdash a,b \dashv$ and $\vdash a \dashv$. An *LR* parser attempting to recognize $\vdash a,b \dashv$ for this grammar would go through the following steps.

Initially, the parser's stack would be empty, and the input symbol would be $\vdash$. This first symbol would now be pushed, or *shifted*, onto the stack. This is not the right hand side of any production in the grammar, so no further action is taken. At this point the stack contains

$$\vdash$$

The next input symbol, "a" is now pushed onto the stack. Immediately, the parser notices that it has seen the right-hand side of one of the productions of G, namely $ELEMENT \rightarrow a$. One element is popped off the stack, and the non-terminal *ELEMENT* is pushed onto it. We say that the parser has *reduced* by this production. The stack contents are now

$$\vdash \text{ELEMENT}$$

Once again, however, a right-hand side is seen on the stack. This time, the parser reduces by the rule $LIST \rightarrow ELEMENT$, leaving the stack as

$$\vdash LIST$$

Now, however, no more right-hand sides can be seen on top of the stack, so the parser reads and shifts the next token, in this case a comma. Again, no right-hand side is seen, so the next token is read and shifted — a "b." The contents of the stack are now

$$\vdash LIST\,,\, b$$

Here the parser can reduce by $ELEMENT \rightarrow b$, and then by $ELEMENT \rightarrow LIST$, leaving the stack as

$$\vdash LIST\,,\, ELEMENT$$

Here, too, is the right-hand side of a production, in particular the production $LIST \rightarrow LIST\,,ELEMENT$. Popping three elements off the stack, and replacing them with *LIST*, we have

$$\vdash LIST$$

Finally, the last input symbol, $\dashv$, is shifted onto the stack, giving

$$\vdash LIST \dashv$$

Now we see the right-hand side of the goal production, $\Sigma \rightarrow \vdash LIST \dashv$. Reducing by this rule, we leave the stack containing only the goal symbol, Σ. This verifies that the string $\vdash a\,,b \dashv$ is indeed generated by grammar G.

If one checks the stack during the above process, one will notice that, at all times, some number of the elements on top of the stack formed the beginning of a right hand side of some production. As soon as this right hand side was completed, a reduction took place. A syntax error has been detected when, if the next symbol were shifted, this property would no longer hold.

1.5. What to Do When You've Missed a Semicolon

Once a syntax error is detected, we must decide how to recover from the error. Three basic options are open to us: halt, repair all errors and execute the resulting repaired code, or merely flag all errors with no attempt at correction.

The first of these, halting the parser, used to be the least attractive of the three. Since only one error is detected per attempted compilation, the process of removing syntax errors from a program would become rather tedious, to say the least. However, certain micro-computer compilers, notably Borland's Turbo Pascal and Turbo C use this approach. They are successful because they find the first error very quickly and automatically invoke an editor so that the programmer is positioned in the file at the point of the error.

The remaining two schemes continue past the first error in the input. They differ primarily in how they attempt to deal with the errors detected. The correction approach takes the attitude that it should make some correction for every error, and attempt to execute the program described by the corrected text, in the hopes of detecting runtine errors. This approach is exemplified by the PL/C compiler of Conway and Wilcox [CoW73]. Most current compilers use a somewhat weakened version of this approach, in that they make corrections in order to continue parsing the input and find more syntax errors, but only execute the resulting text if the errors encountered were minor and easily fixed.

The last approach, that of flagging all errors but making no corrections, is not widely used. This is largely due to the fact that current parsing technology depends on having seen a correct prefix to work properly. Once an error has been detected, we no

longer have such a prefix, and these parsers cannot be restarted in the middle of an input stream. The advantage of this technique is that, since no corrections are made, it is impossible to make a wrong correction. Probably every programmer has had the experience of a compiler "correcting" a syntax error in such a way that even more errors were generated.[3]

1.6. A Guide to this Thesis

In this section, we give a brief outline of the remainder of this document. This will hopefully serve to point out those areas which are of the most immediate interest to the reader. Chapter 2 presents a review of the relevant literature on error repair and recovery. Chapter 3 discusses Earley's [Ear70] parsing algorithm in detail. It also presents Cormack's [Cor87] suffix parser, and gives a motivation for attempting to combine the two into a single recovery scheme. Chapter 4 describes how Earley's algorithm may be modified to become a suffix parser. Chapter 5 presents a method for using the information collected by the modified Earley parser to guide error repair, and describes an initial implementation. Finally, Chapter 6 presents conclusions, and possible directions for future research.

---

[3] These "extra" errors are often referred to as *spurious* errors.

CHAPTER 2

BACKGROUND

2.1. Introduction.

Automated syntax error recovery has been a topic of investigation for some time. An early attempt at an error correcting parser was by Irons [Iro63]. Other attempts have been made by Graham and Rhodes [GrR75], Lyon [Lyo74], James and Partridge [JaP73], Mickunas and Modry [MiM78], and by Graham, Haley, and Joy [GHJ79]. Some of these techniques are discussed below.

2.2. Ad Hoc Methods

Ad hoc methods are those which must be largely hand coded, and/or depend on knowledge of the particular language being compiled. In their PL/C compiler, Conway and Wilcox [CoW73] use transition matrices to guide the parser. Entries in tables corresponding to invalid pairs of tokens point to error routines specifically designed to handle that particular type of error. Indeed, "it was left to the judgement of the implementor . . . what would constitute the most plausible repair in each case." [CoW73] These general routines would either delete the next symbol, insert a symbol in the output string, replace the next symbol with another, or delete the previous symbol.

Another *ad hoc* method, which has been used in Yacc [Joh75], is that of *error* productions. In this form of error recovery, the grammar is augmented with productions of the form

$$A \rightarrow \textbf{error}$$

Here, **error** is a special nonterminal. When a syntax error occurs, the current input

token is replaced by **error**, and the stack is popped until a state is reached which can shift over this new token. Any semantic actions associated with this production are then executed, hopefully making any necessary changes to the state of the parser to recover from the error. Again, as in the PL/C scheme, most of the real work of error recovery is left to the programmer. He must decide not only where to place the error productions, which can often lead to ambiguities in the grammar if not used properly, but he must also write the recovery routines themselves.

In general, *ad hoc* routines tend to handle the anticipated errors very well, but the unanticipated ones poorly, if at all.

2.3. Panic Mode

The simplest, and most widely used, method of error recovery is commonly called *panic mode.* In this form of recovery, the input is scanned until one of a special set of tokens, or *fiducial* [PaK79] symbols (commonly including `end` and `;`) is seen. Then, the stack is popped until this token can be shifted. The biggest drawbacks of this method result from its very simplicity. Namely, none of the available information about the error is used, and a large amount of input can potentially be skipped, none of which will be checked for errors. As well, this method can cause a great many spurious error messages, since the fiducial symbol found may not properly complete the phrase that was being parsed at the point of the error.

Although panic mode is easy to implement and to understand, its limited power and effectiveness make it unsuited for syntax error recovery. The primary advantage of panic mode recovery over the *ad hoc* methods is the fact that panic mode is basically language independent, and could be built into an automated parser generator.

2.4. Graham and Rhodes

Graham and Rhodes [GrR75] propose a method of error recovery based on the notions of *forward* and *backward* moves around the point of the error. Essentially, the attempt to gather information from the context both to the left and right of the point of error detection, and from this deduce the proper repair for the error. Their method is

based on *simple precedence* parsers, however, it has been extended to *LR* parsers by Mickunas and Modry [MiM78].

Error recovery is invoked when an error is detected by the parser. Control returns to the parser when the error state has been corrected. The recovery method has two main phases: the *condensation* phase, and the *correction* phase. The condensation phase is used to gather information from the context of the error — both to the left and to the right. Graham and Rhodes do not retain previously scanned input, so left context amounts to examining the parse stack. The condensed context is used to guide the search for feasible repairs. This differs from many simpler forms of error recovery that merely consider the current state of the parser in determining what recovery action to take.

The condensation phase itself consists of two parts, a *backward* and a *forward* move. Consider the following example[1], taken from [GrR75]:

```
... some more statements ...
write (begin I := 3 end);
```

This statement is illegal in the dialect of Algol used by Graham and Rhodes. The error here is detected when the `begin` is seen by the parser. The stack configuration at this time is

<blockbody> <procedure id> (?

with `begin` as the current input symbol. The *?* represents the point on the stack at which the error was detected. The backward move is now initiated. The purpose of the backward move is to examine the stack to the left of the error, and try to find more reductions to make. In this case, *<procedure id> (* can be reduced to *<procedure head>*, resulting in the modified stack

<blockbody> <procedure head> ?

---

[1] The grammar describing these examples is given in [GrR75], and is also reproduced in the Appendices.

Since no more reductions can be found to the left of the error point, the backward move is finished.

The recovery algorithm now begins the forward move. The forward move attempts to continue parsing beyond the point of the error.[2] In this case, the forward move continues until the stack is

<blockbody> <procedure head> ? <blockbody> <statement> `end`

with current input of `)`. At this point another error is detected in the string, namely that a closing parenthesis cannot follow an `end` token. A second backward move is started, this time from the second error. It modifies the stack to contain

<blockbody> <procedure head> ? <statement>

with the current input symbol as a closing parenthesis. This completes the forward move, and also the condensation phase of the algorithm. Note that the forward move provides essentially unbounded lookahead. In this example, for instance, the text between the `begin` and `end` could have been an arbitrarily long statement, and the final form of the stack would have been the same.

Now that the condensation phase is complete, as much data as possible has been gathered about the left and right contexts of the error. The correction phase now attempts to change the incorrect stack into a stack representing a valid sentence of the language. Often, there is more than one way in which to modify the stack. Graham and Rhodes use a weighted minimum distance measure to determine which repair is "best." Associated with each grammar symbol are two weights: one representing the cost of inserting that symbol into the stack, and one representing the cost of deleting that symbol from the stack. For symbol X, these are represented by $I(X)$ and $D(X)$ respectively. One possible repair for our example is to replace the *<statement>* with a *<expression>*. The cost of this repair is given by $I(<expression>)+D(<statement>)$. After all possible repairs have been examined, the one with the least cost is used, the

[2] This is particularly easy with precedence parsers, which will start parsing at any point in a string, and continue shifting and reducing as long as possible.

stack repaired, and control returned to the parser.

2.5. Graham, Haley, and Joy

Graham, Haley, and Joy [GHJ79] have presented a method for error recovery that is in use in the Berkeley UNIX[3] *pc* Pascal compiler. The method incorporates the ideas of a forward move and weighted insertion and deletion costs, both from Graham and Rhodes, however it also makes use of semantic information and error productions. The method does not repair the source text, in that it does not attempt to convert the (syntactically incorrect) source into a syntactically correct program. Rather, the approach taken merely attempts to continue parsing past the point of the error.

As with Graham and Rhodes, there are two phases to the algorithm: first level, and second level recovery. First level recovery consists of insertion, deletion, or replacement of a single token. This class of error covers a large variety of the more common syntax errors. Second level recovery is invoked whenever first level recovery fails. This second level consists of a refined form of panic mode combined with error productions.

The changes considered by first level recovery are single token changes involving the shiftable tokens in the current parser state. The changes considered are: *insertion* of each possible shift token before the input token causing the error, *deletion* of the input token, and *replacement* of the input token with each possible shift token. First level recovery initially scans ahead in the input a fixed number of tokens (five in the *pc* compiler). These tokens are used for the forward move strategies. Next, each possible repair is attempted, and the parse continued in an attempt to consume as much of this bounded lookahead text as possible. The forward move stops when another error occurs, a reduction indicating an semantic error occurs, or the lookahead is exhausted. This differs from the Graham-Rhodes algorithm which could do a potentially unbounded lookahead during its forward move. Moreover, if the last move of the parser was a shift, the state of the parser is backed up one move, and these actions are

---

[3]UNIX is a trademark of AT&T Bell Laboratories.

repeated. Backing up only over a shifted token eliminates the need to undo reductions, and their associated semantic actions. In order to choose the best repair, a weighting scheme similar to that in the Graham-Rhodes algorithm is used. In addition, each of these scores is multiplied by a factor related to how much of the input was consumed by the repair. The factor is smaller for repairs that proceed farther into the input. Thus, a repair that successfully consumes all five lookahead tokens will be preferable to one whose initial cost is the same, but only consumes two of the lookahead tokens.

Should first level recovery fail, second level recovery is invoked. As noted above, this phase is a combination of simple panic mode with error productions. More specifically, the stack is popped until a state with a shift on an *error* token is reached. Input is then skipped until a token is reached which can either be shifted in this state, or is in a set of predefined "beacons."

In addition to strictly syntactic information, Graham, Haley, and Joy also take advantage of semantic information in deciding which change to make. This eliminates many repairs that are syntactically correct but semantically meaningless. As well, certain syntax errors have been handled by "lifting" them from the grammar, and detecting them via semantic actions. For example, the grammar used for their version of Pascal allows the `label`, `const` `type`, and `var` sections to occur in any order, even though this is not correct in standard Pascal [JeW75], and instead, detect this error through semantic actions. This does, however, require the error recovery routines to perform semantic checks to eliminate proposed corrections that may otherwise be syntactically correct.

The key limitation of this method is the restriction to single token corrections. The use of a bounded lookahead greatly increases the ability of the algorithm to choose the correct single token repair when one can be found, however, if this fails, the algorithm falls back on a relatively primitive scheme of merely skipping input until some synchronizing token is seen.

2.6. Lyon's Method

Lyon's algorithm [Lyo74] differs from those discussed above in that, instead of making a *local* correction, it attempts to find a *least-errors* correction of the input string. In this discussion, *least-errors* will mean the fewest number of insertions, deletions, or replacements needed to change the original input into a correct program. Such a correction is also called a *minimum distance* correction.

Lyon's algorithm is based on Earley's parser [Ear70][4] combined with *dynamic programming* to achieve a minimum distance correction of the input string. The key to Lyon's algorithm is the observation that, if $Z \rightarrow XY$ is a production in some context-free grammar, then any string derived from Z using this production will have a number of errors in it equal to the sum of the number of errors in the substrings derived from X and Y. This is critical, because it allows Lyon to find a minimum distance correction for the substrings derived from Y and Z and then be guaranteed that they together form a minimum distance correction for the substring derived from Z.

The Earley parser associates a set of items with each position in the input string, each set representing the state of the parser at that particular token. These items, normally, consist of a production, a *dot* representing how far that particular production has been matched, and a pointer back to the state where that production began to be matched. A typical item may look like

$$[Z \rightarrow \alpha \cdot \beta, 5]$$

indicating that the parser has seen α, and is waiting to see β, and started matching this instance of Z in state number five. Lyon augments this information with a count of the number of corrections that have been made thus far in recognizing this production. Thus, a typical item in Lyon's parser looks like

$$[Z \rightarrow \alpha \cdot \beta, 5, 3]$$

In addition to the same information represented by the standard Earley parser, this item

---

[4] For a detailed description of Earley's parser, see Chapter 3.

states that three corrections were necessary to match α successfully.

Lyon's algorithm consumes the input a token at a time, constructing a set of items representing the current state of the parser. As in Earley's algorithm, there are three operations which may be applied to a given item in a state, each of which may result in new items being added to that state, or to its successor. These three operations are the *scanner*, *predictor*, and the *completer*. The predictor, however, is identical to that used by Earley, and as such, is not discussed here. The phrase "adding an item to a state" has a rather specialized meaning in the context of this discussion. In particular, it is possible for the following situation to arise: an item of the form $I_1=[Z \rightarrow \alpha \cdot \beta, p, k]$ being added to a state already containing an item of the form $I_2=[Z \rightarrow \alpha \cdot \beta, p, k']$. In this case, the precise action taken depends on the values of k and k'. If $k<k'$, i.e., the new item has undergone fewer corrections than the old item, then I_1 replaces I_2. If, however, $k>k'$, in which case the old item represents the fewer corrections, the new item is simply discarded. When $k=k'$, there is nothing to choose between the two items *syntactically*,[5] so either may be used. Lyon chooses to use the original item.

Before starting, Lyon's algorithm initializes each of $n+1$ (where n is the length of the input plus an always-correct end-of-file symbol) states. State S_i is initialized to contain items of the form $[Z \rightarrow \cdot \beta, i, 0]$, for all productions Z. This effectively attempts to parse all possible productions of the grammar from all points in the input. This is necessary to allow for the possibility that some arbitrary prefix of the input may need to be deleted.

For each state, S_i, beginning with S_1, the *scanner* is invoked. Let t_i represent the i^{th} token of the input. Then, for each item in S_i of the form $[Z \rightarrow \alpha \cdot c\, \beta, p, k]$, c a terminal symbol, the scanner attempts the four actions below:

(1) If $c=t_i$, then add, as discussed above, $[Z \rightarrow \alpha t_i \cdot \beta, p, k]$ to S_{i+1}. This is a *perfect match*, and as such, does not increase the error count of the state added to S_{i+1}.

---

[5] There may certainly be *semantic* reasons to choose one item over the other, however Lyon does not deal with this issue.

(2) If $c \neq t_i$, then add $[Z \rightarrow \alpha c \cdot \beta, p, k+1]$ to S_{i+1}. This represents a *mutation* error, i.e., t_i should have been c.

(3) Add $[Z \rightarrow \alpha c \cdot \beta, p, k+1]$ to S_{i+1}. This *deletion* hypothesis assumes that c was incorrectly deleted from the input.

(4) Add $[Z \rightarrow \alpha \cdot c\, \beta, p, k+1]$ to S_{i+1}. This *insertion* hypothesis assumes that t_i should not have been in the input.

These four hypotheses, between them, cover all possible errors that could have occurred at t_i. All must be tried, since it is impossible to predict beforehand which one of them actually did occur.

If, for example, state S_i contained item $[Z \rightarrow \cdot a, i, 0]$, then after the scanner had been applied to this state, one or more of the states $[Z \rightarrow \cdot a, i+1, 1]$, via the insertion hypothesis, $[Z \rightarrow a \cdot, i+1, 1]$, via the mutation hypothesis, $[Z \rightarrow a \cdot, i+1, 0]$, via the perfect match hypothesis, and $[Z \rightarrow a \cdot, i, 1]$, via the deletion hypothesis.

Once the scanner has been applied to each state in S_i, the *completer* comes into play. The completer may be applied to items of the form $[Z \rightarrow \beta \cdot, p, k]$. Such items are called *final*, and indicate that a complete phrase derived from Z has been recognized, starting st state S_p, extending to the current state, and having required k corrections. Essentially, the completer algorithm examines an item $[Z \rightarrow \beta \cdot, p, k]$, and then searches S_p for an item of the form $[Y \rightarrow \alpha \cdot Z\, \beta, p', k']$. A new item is then added to the current state: $[Y \rightarrow \alpha Z \cdot \beta, p', k+k']$. Two constraints, however, are required on the order in which final items are considered for completion:

(1) Items should be considered in decreasing order of p. This prevents later completions from interfering with earlier ones, since they will refer to points earlier in the parse.

(2) For any given value of p, items should be considered in *increasing* order of k. This minimizes the amount of work that needs to be done during the completion process by eliminating the possibility of having to undo earlier work.

When the algorithm reaches the end of the string, it examines state S_{i+1}, to find the item $[\Sigma \rightarrow S|\dashv,\cdot,1,k]$. Here, k is the minimum distance of this string from a correct string of the language.

Lyon's algorithm runs in time $O(n^3)$. This is somewhat surprising, at first glance, since Earley's algorithm is only $O(n^3)$ for ambiguous grammars. However, upon closer inspection of Lyon's approach, one sees that he is effectively transforming the original grammar to one containing all possible single token insertions, deletions, and replacements. Such a grammar is highly ambiguous, and accounts for the cubic time bound. This is clearly too slow for practical use. As Lyon points out, however, it may be possible to speed the process up by precomputing as much of the information as possible for a particular grammar. A more serious flaw may be the focus on minimum distance corrections. It is possible, indeed, probable, that many minimum distance corrections do not correspond to what a programmar would consider *natural*. For example, consider the Algol 60 statement

```
i := j k / l m;
```

The two obvious syntax errors here are missing operators between `j` and `k`, and `l` and `m`. The minimum distance correction, however, is the statement

```
comment i := j k / l m;
```

It is highly unlikely that this is what the programmer had in mind. A possible approach to overcoming this defect is to assign, as Graham and Rhodes did, insertion and deletion weights to each token, and use a weighted minimum distance instead.

2.7. Sippu and Soisalon-Soininen

Sippu and Soisalon-Soininen [SiS] describe a method of syntax error correction based on the *phrase level recovery* of Lenius [Len70]. For Sippu and Soisalon-Soininen, a *configuration* of an *LR* parser is a string of the form

$$q_0 q_1 \cdots q_m | a_0 a_1 \cdots a_n$$

The substring to the left of the vertical bar represents the contents of the parsing stack

with q_m at the top. The sequence $q_0 \cdots q_m$ corresponds to a path through the *LR* parsing machine starting from state q_0. The substring $a_0 \cdots a_n$ represents the remainder of the input, with a_0 being the current input symbol and a_n being the end marker. A configuration in which no parsing action is possible is termed an *error configuration.*

The essential idea behind phrase level recovery is the isolation in a parser configuration of an *error phrase,* which can be replaced by a suitable *reduction goal.* In other words, if

$$q_0 \cdots q_m | a_0 \cdots a_n$$

is an error configuration, then a substring

$$q_{i+1} \cdots q_m | a_0 \cdots a_{j-1}$$

is said to be an *error phrase* if we can find a state q_A such that q_A is the successor on A from q_i, and that symbol a_j is acceptable at q_A. The effect of phrase level recovery is to replace the error phrase above by the string $q_A|$, giving the non-error configuration

$$q_0 \cdots q_i q_A | a_j \cdots a_n$$

It should be noted that either the stack portion, or the input portion, or both, of the error phrase can be empty. We call q_i, A, q_A, and a_j the *recovery state, (reduction) goal, goal state,* and *recovery symbol,* respectively.

Two further concepts defined by Sippu and Soisalon-Soininen are *important* reduction goals, and *feasible* reduction goals. A reduction goal A of an error phrase $\beta|x$ is said to be *important* if there is no other reduction goal B which can derive A through only unit productions. This essentially ignores reduction goals like *if_statement* or *while_statement* in favour of *statement*, since the former two will eventually reduce the later, in any case. The idea of *feasible* reduction goals arises from the common assumption (made by Sippu and Soisalon-Soininen) that the input already parsed onto the stack is correct. Under this assumption, it makes sense to require the reduction goal chosen to be able to derive the stack portion of the error phrase. Precisely, a reduction goal A of error phrase $\beta|x$ is *feasible* if A can derive some terminal string with prefix $\overline{\beta}$.

Here, $\overline{\beta}$ is the string of grammar symbols represented by the states of the stack. In practice, Sippu and Soisalon-Soininen use *weak feasibility*, which requires only that the reduction goal be able to derive the first stack symbol of the error phrase. According to Sippu and Soisalon-Soininen, situations where weakly feasible reduction goals are not truly feasible do not occur very often.

Sippu and Soisalon-Soininen's recovery routine works by finding an error phrase $\beta|x$ with exactly one important, weakly feasible reduction goal A, and then replacing $\beta|x$ with $A|$. The search for $\beta|x$ begins with the shortest possible error phrase, that consisting only of the vertical bar, and works through larger and larger segments of the error configuration. Clearly, the exact order in which the error phrases are examined is crucial in determining the repair made. Sippu and Soisalon-Soininen require that the search order be such that no potential error phrase is examined before all of its subphrases are examined. This still leaves considerable leeway in the precise order in which error phrases are examined, however, Sippu and Soisalon-Soininen have found that an order which consumes stack states slightly faster than input tokens gives the best results. It is possible that no feasible repair can be found. In this case, the top state is popped off the stack, and phrase level recovery is attempted again.

Phrase level recovery, as described above, can be unduly influenced by the form of the grammar. For example, if the rules

stmt_list → *stmt_list* ; *stmt*

stmt_list → *stmt*

are used to generate statement lists in Pascal, then a missing semcolon will probably result in the deletion of the entire following statement. If, however, the grammar looked like

stmt_list → *stmt_list* <;> *stmt*

stmt_list → *stmt*

<;> → ;

then the missing semicolon could be inserted, which is definitely a better repair. It is also possible to modify the grammar so that phrase level recovery can handle deletions.

Rather than make these modification directly to the grammar, their effect can be achieved by allowing not only non-terminals as reduction goals, but terminal, and empty strings as well. Allowing these extended reduction goals, however, greatly increases the number of possible recovery actions, including undesirable ones such as replacing large portions of the stack and input with a single terminal symbol. Sippu and Soisalon-Soininen restrict these *local corrections* so that only single-token insertions, deletions, and replacements are allowed. Three costs are associated with each token: an insertion cost, and a deletion cost, in the fashion of Graham and Rhodes, and a *safety* cost, which is a measure of reliability of the correction achieved using this terminal.

In practice, this algorithm performs well. In a collection of erroneous Pascal programs [RiD78], 36.3% of recovery actions were rated excellent, 31.0% good, 22.6% fair, and 10.1% poor. Even with this performance, however, approximately 1.4 error messages were produced for each "actual" error in the sample.

2.8. Conclusions

Each of the methods discussed above has its advantages and disadvantages. Perhaps the single disadvantage common to each is that in every case it is possible for the repair action taken to cause spurious syntax errors later in the parse. Often, these spurious errors occur in cascades, which obscure the legitimate error messages.

CHAPTER 3

EARLEY'S PARSER and CORMACK'S SUFFIX PARSER

3.1. Introduction

The Earley parser [Ear70] is a general parsing algorithm for context free grammars. The algorithm is $O(n^3)$ in the general case, but, for certain subclasses of grammars, becomes $O(n^2)$ or even $O(n)$. Notably, the class of grammars for which the algorithm runs in linear time includes the $LR(k)$ grammars.

Cormack's suffix parser [Cor87] is a generalization of the standard $LR(k)$ generator algorithm which will parse any suffix string of the original grammar. In other words, Cormack's parser will recognize any string that could be generated by deleting an arbitrary number of tokens from the beginning of syntactically correct program. This is useful for error recovery, because it allows the parser to restart easily after an error has occurred, and continue looking for other errors.

3.2. Earley's Algorithm

Earley's algorithm scans an input string $\omega = w_1, \ldots, w_n$ from left to right, and forms a sequence of sets $S_0, \ldots, S_n$. Each set S_i, $i \geq 0$, contains zero or more *items*, and represents the current state of the parse after reading w_i. We shall refer to these sets as *states*. Each item in set S_i represents three things:

(a) a production $p \in P$ in the process of being recognized,

(b) how much of p has already been seen, and

(c) a *backpointer* to the state S_j, $j \leq i$, in which we first started scanning for this particular production.

Consider the grammar described in Figure 1.1. The item $[E \rightarrow E+\cdot T,5]$ indicates that the parser is in the process of attempting to recognize an E, by the second rule of the grammar, and thus far has recognized an E and a +. The $\cdot$, called the *dot*, is a used to denote how far along in the production recognition has progressed. The second part of the item, the 5, indicates that the parser started recognizing this instance of an E in state S_5.

In general, Earley's algorithm applies one of three operations to each item in state S_i, depending on the form of the particular item. These operations may add states either to S_{i+1} or to S_i itself. If new items are added to S_i, these operations are also applied to each of them.

The simplest of the operators is the *scanner*[1]. The scanner may be applied to an item in state S_i if the dot is before a terminal. For example, consider again grammar from Figure 1.1, and input string $\vdash x \dashv$. When the parser starts recognizing this sentence, S_0 contains the single item

$$\Sigma \rightarrow \cdot \vdash E \dashv, 0$$

This item indicates that we have not yet recognized any portion of a valid arithmetic expression. If the current input symbol matches the terminal after the dot, in this case a \$, the scanner creates a new item

$$\Sigma \rightarrow \vdash \cdot E \dashv, 0 \qquad (1)$$

and adds that item to state S_1. More generally, the scanner advances the position of the dot by one position in the production, and adds the new item to the next state. This exhausts the items in S_0, so we are finished here.

The algorithm now considers S_1, which contains the single item (1) above. The scanner is not applicable, since the dot is before a nonterminal. The *predictor*, however, is. The predictor is applicable whenever the dot is just before a nonterminal Z. It

---

[1] Note that this term has no relationship whatsoever with the more common usage of the term *scanner* to mean lexical analyzer.

adds one new item to the current state for each production in the grammar with Z as its left hand side. The backpointer of each item added is set to the number of the current state, to indicate that we started looking for an instance of Z at this place, and the dot is placed at the beginning of the right hand side.

In our example, we apply the predictor to (1), adding two new items

$$E \rightarrow \cdot E+T, 1 \tag{2}$$

$$E \rightarrow \cdot T, 1 \tag{3}$$

to S_1. Having processed item (1), we continue and process item (2). Again, the predictor is applicable. However, applying the predictor here simply adds duplicates of (2) and (3) to S_1, which are ignored. The algorithm moves on to (3). Once more, the predictor is applicable. Now, however, the dot is before a T, and we add the following two items to S_1:

$$T \rightarrow \cdot T*F, 1 \tag{4}$$

$$T \rightarrow \cdot F, 1 \tag{5}$$

The algorithm applies the predictor to (4), which results in the same two items being added a second time. When the predictor is applied to (5), the item

$$F \rightarrow \cdot x, 1 \tag{6}$$

is added to S_1. At this point, the predictor is no longer applicable to any unprocessed item. The scanner, however, can be applied to (6). Since the next input character is x, the scanner can be applied successfully, and S_2 becomes

$$F \rightarrow x \cdot, 1 \tag{7}$$

Neither the predictor nor the scanner is applicable to this item. Instead, the *completer* may now be used. The completer is used only for those items in which the dot is at the far right end of the production. Such items are also called *final*, or *complete*, items. It examines the state referred to by the backpointer of the completed item, in this case S_1, and adds all items in S_1 in which the dot is before F to the current state, S_2, after

moving the dot forward one position. Applying the complete to (7), we add

$$F \rightarrow T\cdot, 1 \tag{8}$$

to S_2. This item is also final, so the completer is applicable here as well. This adds the states

$$\begin{aligned} T &\rightarrow T\cdot * F, 1 \\ E &\rightarrow T\cdot, 1 \end{aligned} \tag{9}$$

to S_2. The latter one of these is also final, and so the complete is applied to it as well. This adds the states

$$\begin{aligned} E &\rightarrow E\cdot + T, 1 \\ \Sigma &\rightarrow \vdash E\cdot \dashv, 0 \end{aligned} \tag{10}$$

Neither the completer or the predictor can be further applied to state S_2. The only applicable operator is the scanner. Since the current input symbol is now $\dashv$, state S_3 becomes

$$\Sigma \rightarrow \vdash E \dashv\cdot, 0$$

But, this item indicates that we have successfully matched Σ, the goal symbol of the grammar. Since we have exhausted the input, the string is clearly a member of the language generated by grammar of Figure 1.1.

Earley's algorithm, as described above, implements a *recognizer* for a language. This differs from a parser in that a recognizer merely answers the question "Is string x a member of language L?" without generating a derivation tree for the x. In order to convert the recognizer into an parser, a few additions to the algorithm described above are required.

The major change takes place in the completer. Every time an item is completed, all the items it causes to be added to the current state should have a pointer from the terminal symbol that was just passed over to the item just completed. For example, if during the process of completing $I_1 = [X \rightarrow \alpha\cdot, j]$ an item of the form $I_2 = [Z \rightarrow \beta X\cdot\gamma, k]$ is

added to the current state, the completer should set a pointer from the instance of X in I_2 to I_1. If certain completions cause the same item to be added several times, one pointer is required for each completion. Finally, when the final state is reached, with the only item being $[\Sigma \rightarrow \vdash S \dashv \bullet, 0]$, a factored parse tree of all parses for the input string will be hanging off of S. The parse which would be produced for the string $\vdash x+x \dashv$ is shown in Figure 3.1.

3.3. Cormack's Suffix Parser

Cormack's suffix parser [Cor87] is a modification of the standard $LR(k)$ parser construction method. Like all LR parsers, it maintains a stack of states, representing the input seen so far. The suffix parser for a grammar G accepts string ω if there exists a string γ such that $\gamma\omega$ is in $L(G)$.

Consider the grammar below:

$$\Sigma \rightarrow A$$
$$A \rightarrow \lambda$$
$$A \rightarrow (A)$$

This grammar generates the language $L(G)=\{(^k)^k | k \geq 0\}$. Let $\omega=())$. It is clear that ω is a suffix of some string in $L(G)$, namely, $(())$. Before reading the first token, the parser has no information about where it may need to begin parsing. Because of this, the initial state of the parser must allow for all possible terminal symbols. The initial state, therefore, consists of the two items

$$[A \rightarrow \cdots \bullet (A)]$$
$$[A \rightarrow \cdots \bullet)]$$

Items of the form $[Z \rightarrow \cdots \bullet \beta]$ where $Z \rightarrow \alpha\beta$ is a production, are called *suffix items*, and indicate that some initial part of α may never have been seen. More generally, for an arbitrary grammar $G=(N,T,P,\Sigma)$, the initial state of the suffix parser will be the set of all suffix items of the form $[Z \rightarrow \cdots \bullet t\beta]$ where $Z \rightarrow \alpha t \beta$ is in P, and with t ranging over all terminal symbols.

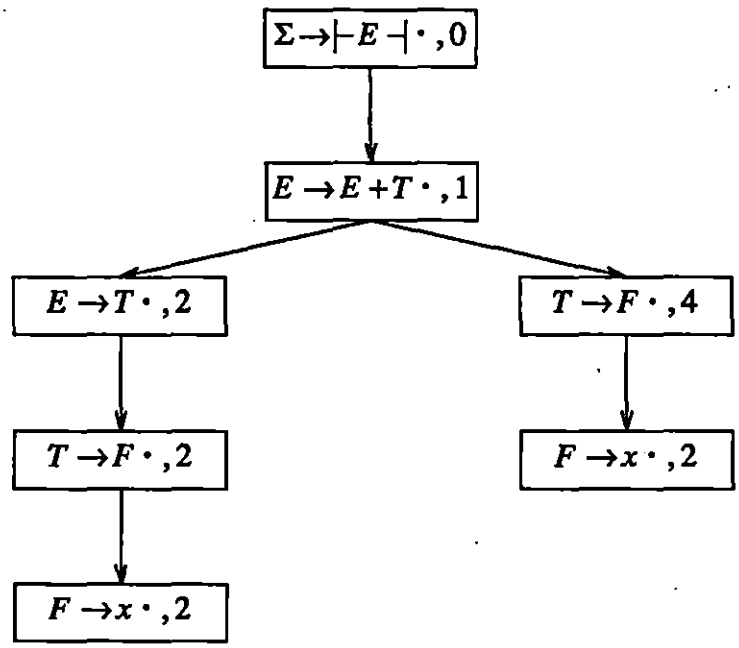

Figure 3.1. Parse Tree Generated by Earley's Algorithm

Now, the parser reads the first token of the input, an opening parenthesis. Upon reading this token, it pushes a new state onto the stack. Since the current token is a "(", we are clearly recognizing the first, rather than the second, of the two items from the initial state. Thus, the next state initially contains

$$[A \rightarrow \cdots \cdot A)]$$

However, in this state, the dot is before a non-terminal. We add the items

$$[A \rightarrow \lambda\cdot], \text{ and}$$
$$[A \rightarrow \cdot(A)]$$

to the current state. Had the dot been before a non-terminal in either of these new items, the process would have been continued until no new items could be added to this state.

The next input token is a closing parenthesis, ')'. No item in the current state has the dot before this non-terminal. However, the second item, $[A \rightarrow \lambda\cdot]$, indicates that the parser can successfully recognize an A as the empty string, λ. The parser therefore pops the top $|\lambda|$ states off the stack, and acts as though the input symbol is now an A. In this case, no states are popped since λ is the empty string, and $|\lambda|$ is zero.

Since there is an item in this state with the dot before an A, yet another new state is pushed onto the stack. This state contains the single suffix item

$$[A \rightarrow \cdots\cdot)]$$

The current input token, still '(', is now usable. Yet another new state is pushed on the stack, initially containing

$$[A \rightarrow \cdots\cdot]$$

An item of this form indicates that the parser has successfully matched a suffix of an A, in this case the string (). Since the parser doesn't know how much of the production it has seen and how much was missing, it cannot reduce this rule by popping states from the stack. Instead, for every production $Z \rightarrow \alpha A \beta$, a suffix item $[Z \rightarrow \cdots\beta]$ is added to the current state. In our example, this results in the following new items:

$$[\Sigma \rightarrow \cdots\cdot]$$
$$[A \rightarrow \cdots\cdot)]$$

Once again, the parser reads a new input token, in this case another closing parenthesis. Another new state is pushed on the stack. Again, this state contains the items

$[A \rightarrow \cdots\cdot]$
$[\Sigma \rightarrow \cdots\cdot]$
$[A \rightarrow \cdots\cdot)]$

This state is identical to the previous. Indeed, it is clear that any sequence of closing parentheses will simply result in many copies of the same state being pushed on the stack. This state also contains the item $[\Sigma \rightarrow \cdots\cdot]$. This item indicates that we have seen a string which may be a suffix of Σ, the goal of the grammar. If any state containing this item is on top of the stack when the input is exhausted, the input string was indeed a suffix of some member of $L(G)$. In our example, this is the case. Note that, unlike *LR* parsers, the suffix parser can, and in general, will, leave information on the stack.

As we have presented it here, the suffix parser algorithm is continually building new states as it reads the input. In practice, this need not be done. This fact depends upon the observation that the number of possible suffix and non-suffix items is finite, which implies that the number of possible sets of such items is also finite. Since each state constructed by the parser is a set of suffix and non-suffix items, it follows that the number of possible states is also finite. It is therefore possible to precompute all the states and transitions from them, and then simply push state numbers onto the stack, instead of the sets of items themselves.

The algorithm for constructing the states of the parser is basically that used for constructing the *LR* parsers, with the necessary extensions to deal with suffix items. Let $G=(N,T,P,\Sigma)$ be a context-free grammar. Then, the start state of the suffix parser for G is the set of items of the form $[Z \rightarrow \cdots\cdot t\beta]$, where $Z \rightarrow \alpha t \beta$ is in P, and t is a terminal symbol. Now, every state I has an outgoing transition for each symbol, terminal or non-terminal, which is preceded by a dot in some item of I. Let x be the symbol in question, where $x \in (N \bigcup T)$ The state reached by this transition, I', is computed by repeatedly applying the following rules until no new items are added to I'.

(1) For each item of the form $[A \rightarrow \alpha\cdot x\beta]$ in I, add $[A \rightarrow \alpha x\cdot\beta]$ to I'.

(2) For each item of the form $[A \rightarrow \cdots \cdot x\,\beta]$ in I, add $[A \rightarrow \cdots \cdot \beta]$ to I'.

(3) For each item of the form $[A \rightarrow \cdots \cdot C\,\beta]$ in I', where C is a non-terminal, add all items of the form $[C \rightarrow \cdot \beta]$, where $C \rightarrow \beta$ is a production of G.

(4) For each item of the form $[A \rightarrow \alpha \cdot C\,\beta]$ in I', where C is a non-terminal, add all items of the form $[C \rightarrow \cdot \beta]$, where $C \rightarrow \beta$ is a production of G.

(5) For each item of the form $[A \rightarrow \cdots \cdot]$ in I', add items of the form $[Z \rightarrow \cdots \cdot \beta]$, where $Z \rightarrow \alpha A\,\beta$ is a production of G.

Keep generating new states until no more distinct states can be generated. These are then the states of the suffix parser for grammar G. For the original grammar of this section, this process generates the following seven states:

I_0: $[A \rightarrow \cdots \cdot (A)]$
$[A \rightarrow \cdots \cdot)]$

I_1: $[A \rightarrow \cdots \cdot A)]$
$[A \rightarrow \lambda \cdot]$
$[A \rightarrow \cdot (A)]$

I_2: $[A \rightarrow \cdots \cdot)]$

I_3: $[A \rightarrow \cdots \cdot]$
$[\Sigma \rightarrow \cdots \cdot]$
$[A \rightarrow \cdots \cdot)]$

I_4: $[A \rightarrow (\cdot A)]$
$[A \rightarrow \lambda \cdot]$
$[A \rightarrow \cdot (A)]$

I_5: $[A \rightarrow (A \cdot)]$

I_6: $[A \rightarrow (A) \cdot]$

Readers familiar with *LR* parsers will observe that the final three states are also found in the *LR* parser for this grammar. In fact, this must be the case, since the set of all suffixes of language L includes all strings of L. The transitions between these states are shown in Figure 3.2 below.

The parser just constructed is called a (1)*LR*(0) parser by Cormack. More generally, a $(k)LR(m)$ parser uses k symbols of initial context to determine what to do from the start state, and m symbols of lookahead in deciding all other parsing actions. Upon closer examination, the parser that has just been constructed is non-deterministic since in both states I_1 and I_4 the automaton may either reduce by $A \rightarrow \lambda$ or shift into a

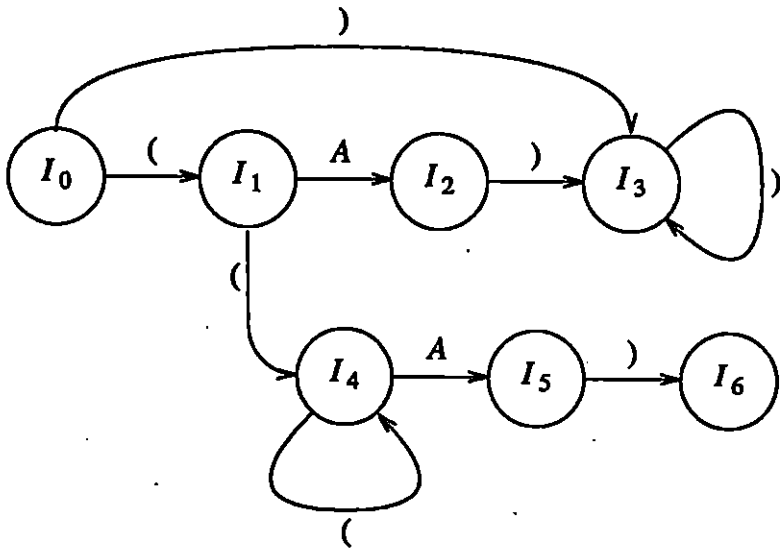

Figure 3.2. Transition Diagram of Suffix Parser

new state when the input is an opening parenthesis. This particular conflict can be resolved by making use of the follow set for A. This is analagous to the construction of the *SLR* (1) parsers from *LR* (0) parsers [DeR71], and is termed a (1)*SLR* (1) parser by Cormack.

Cormack has implemented a (1)*SLR* (1) parser generator, and used it to create a suffix parser for Pascal. The resulting suffix parser has more than twice as many states as the corresponding *SLR* (1) parser (1000 as opposed to 400)[2]. Cormack argues that this is still not "too" big, and this seems a reasonable position. A somewhat more

[2] The grammars used for the *SLR* (1) and the (1)*SLR* (1) parsers are, of course, different.

serious problem arises when two distinct groups of productions generate fragments that can be confused with one another. This situation leads to conflicts in the parsing automaton. The difficulty can be removed, however, by appropriately rewriting the grammar. This rewriting, while it preserves the phrase structure of the grammar, unfortunately leaves the new grammar larger than the original, and usually harder to understand.

3.4. Conclusions

Earley's algorithm provides a linear time parser for any $LR(k)$ grammars but cannot, in the form described here, restart after an error in the input string. Cormack's suffix parser provides the ability to restart after an error, but at the cost of restricting the possible input grammars. In the following chapter, we present an algorithm that combines the positive aspects of both methods, while having the drawbacks of neither.

CHAPTER 4

A SUFFIX PARSER BASED ON EARLEY'S PARSER

4.1. Introduction

In the previous chapter, we described Earley's parsing algorithm. Now, we describe our extensions to Earley's parser which change it into a suffix parser. This will be done via a method similar to that of Cormack, also described in the previous chapter. Extending the parser in this manner allows it to avoid making any assumptions about the nature of the error until later when (presumably) more complete information is available.

4.2. Error Detection

Earley's algorithm works by carrying along all possible parses simultaneously. In this context, a syntax error occurs when there are no more possible parses to pursue. When this happens, the next set of items which the algorithm should process will be empty. In order to continue parsing without referring to previous context, the parser must be able to deal with matching *suffixes* of productions. This, however, is exactly what is achieved by the suffix items of Cormack's suffix parser. The desired solution, then, is to modify Earley's algorithm to use these suffix items.

In the Earley parser, non-suffix items have the form

$$[Z \rightarrow \alpha \cdot \beta, j]$$

where $Z \rightarrow \alpha\beta$ is a production of some grammar, the $\cdot$ indicates how much of this production has been recognized, and j is a backpointer to the state in which the parse started looking for this instance of a Z. A suffix item

$$[Z \rightarrow \cdots \beta \cdot \gamma, j]$$

indicates that the parser has missed the initial portion of production $Z \rightarrow \alpha\beta\gamma$, presumably because a syntax error occurred somewhere in a portion of the input derived from α. As in the non-suffix items, the position of the dot indicates that we have since successfully recognized text derivable from β, and are now ready to recognize text derivable from α. The value of the backpointer j is undefined in a suffix item, since the beginning of the string derived from this production must occur before the last syntax error. Like normal items, suffix items with the dot at the rightmost end of the production are called *final*, or *complete*, suffix items.

Unlike Cormack's algorithm, suffix items in the modified Earley parser are introduced only after a syntax error has been detected. Let the input token at which the syntax error was discovered be t. Immediately after the error is discovered, the next set of states is empty. It is necessary to place some new items in this state so that parsing may continue. Since these items cannot refer to prior context, they must be suffix items. In general, the set of items which the parser starts after a syntax error at token t are those items of the form $[Z \rightarrow \cdots \cdot t\beta, -]$. This set of items indicates that the algorithm must be prepared to see the suffix of any production $Z \rightarrow \alpha t \beta$. Since it is now sure to be successfully used, the parser will use the t as the input token for this set of items. Once suffix items have been introduced into the sets of items, it is necessary to extend to operations of the Earley parser to handle them properly.

The primitive operation of the Earley parser is adding an item to a set of items. In the unmodified algorithm, this operation is quite straightforward, ensuring only that the same item does not occur twice in the same state. With the introduction of suffix items, a slightly more complex definition becomes necessary. When attempting to add a suffix item $[Z \rightarrow \cdots \beta \cdot \gamma, -]$ to a state, it should not be added if a suffix item $[Z \rightarrow \cdots \alpha\beta \cdot \gamma, -]$ already exists in the state. This restriction is in place because the latter item logically implies the former, and no new information can be gained by introducing the new item into the set.

Given this modified definition of "adding an item to a state," the three (higher level) operations of the Earley parser may be discussed. These three operations are the *scanner*, *predictor*, and *completer*. Of these three, only the completer behaves differently on suffix items than on non-suffix items. The completer is applicable to any final item, whether it be a suffix of non-suffix item. The difference lies in how to determine what new items to add the the current state. For a non-suffix item, say $[E \rightarrow E+T\cdot, j]$, the completer examines state S_j for items of the form $[Z \rightarrow \alpha\cdot E\,\beta, k]$. If such an item is found, the dot is moved forward over the E, and the item $[Z \rightarrow \alpha E\cdot\beta, k]$ is added to the current state. Now consider the suffix item $[E \rightarrow \cdots +T\cdot, -]$. Here, the backpointer is undefined, so the parser cannot examine some prior state to determine what new items to add. Instead, for every production of the form $Z \rightarrow \alpha E\,\beta$, a suffix item $[Z \rightarrow \cdots E\cdot\beta, -]$ is added to the current state.

These modifications allow the Earley parser to recover from any syntax error and continue parsing the remainder of the input. More importantly, since the method by which the parsing is continued is independent of prior context, no *spurious* syntax errors will be detected. This in itself is a useful accomplishment.

As an example of the modified Earley parser, consider once again grammar AE, given in chapter 1, and the invalid input string $\vdash + x \dashv$.

When the modified Earley parser, state S_0 contains the single item

$$[\Sigma \rightarrow \cdot \vdash E \dashv, 0]$$

and the current input token is $\vdash$. The scanner is applicable in this situation, since the current input token matches the symbol after the dot in this item. The scanner proceeds to add $[\Sigma \rightarrow \vdash\cdot E \dashv, 0]$ to S_1. This finishes processing for state S_0.

The parser now begins to work with S_1. Initially, the current input token is +, and S_1 consists of only

$$[\Sigma \rightarrow \vdash\cdot E \dashv, 0]$$

Here, the predictor may be used, because the dot precedes a non-terminal. The predictor adds

$$[E \rightarrow \cdot E+T,1]$$

$$[E \rightarrow \cdot T,1]$$

$$[T \rightarrow \cdot T*F,1]$$

$$[T \rightarrow \cdot F,1]$$

and

$$[F \rightarrow \cdot x,1]$$

to S_1. After these states have been added, the predictor is no longer usable on this state. As well, neither the predictor nor the scanner are applicable.

The algorithm now moves on the state S_2. Unfortunately, S_2 is empty, since no items were added to it during the processing of S_1. This tells us that a syntax error occurred at the current input token. In order to continue parsing, the algorithm examines the grammar for all productions containing a +. The only such production in this grammar is $E \rightarrow E+T$. The item

$$[E \rightarrow (E)\cdot+T,-]$$

is placed in state S_2, and normal processing continues. Immediately, the scanner can be used, since the current input token matches the symbol after the dot. This adds the item $[E \rightarrow (E)+\cdot T,-]$ to S_3, and completes handling of S_2.

State S_3 begins with the single item

$$[E \rightarrow (E)+\cdot T,-]$$

The predictor can once again be used, adding

$$[T \rightarrow \cdot T*F,3]$$

$$[T \rightarrow \cdot F,3]$$

and

$$[F \rightarrow \cdot x,3]$$

Now the scanner can be applied to the last item, causing $[F \rightarrow x\bullet,3]$ to be added to S_5.

Once processing starts on state S_5, the completer can be applied to the first item,

$$[F \rightarrow x\bullet,3].$$

Since this item is not a suffix item, state S_3 is searched for items to bring forward to the current state. This search adds

$$[T \rightarrow F\bullet,3]$$

The completer can be applied once again, this time adding

$$[T \rightarrow T\bullet * F,3]$$

and

$$[E \rightarrow (E)+T\bullet,-]$$

Of these two states, the first is ignored, because the current input token, $\dashv$, does not match the symbol following the dot. The second item is final, and thus can be completed. This is a suffix item, however. Rather than examining some prior state in the parse, the grammar is searched for productions with an E in their right hand sides. This adds items

$$[E \rightarrow \cdots E\bullet + T,-]$$

and

$$[\Sigma \rightarrow (\vdash)E\bullet\dashv,-]$$

The first of these is ignored, and the scanner can be used on the other.

Applying the scanner to the last item in S_4 causes

$$[\Sigma \rightarrow (\vdash)E\dashv\bullet,-]$$

to be added to state S_5. This item indicates that we have successfully parsed the remainder of the string.

A suffix parser will *not* detect all syntax errors. Consider the (incorrect) Pascal program below:

```
program fred(input, output);
var a, b: integer;
begin
    if a = b then begin
        a := b b;
end.
```

There are two syntax errors in this program — a missing operator in `a := b b`, and a missing `end` for the `begin` of the if statement. A suffix parser will only report the first of these errors. To see why, note that the fragment `b ; end.` is a possible suffix of a Pascal program. In particular, this fragment is identical to the text of the example after the syntax error. The suffix parser effectively discards all knowledge of what had gone before the point of the error in order to avoid making any assumptions about its correctness. This same assumption, however, causes such "nesting" errors to be missed.

It is relatively straightforward to convert the recognizer just described into a parser. As in the unmodified Earley parser, each completion builds a node of the parse tree. The only difference is that when the parser is finished, there may be a forest of subtrees, rather than one single tree. This will occur whenever the input string has a syntax error. The subtrees represent those parts of the input that were correctly parsed, independent of the errors. How to take these partial parse trees and build a complete parse tree that (hopefully) corresponds to the "correct" version of the program is the other extension to Earley's algorithm.

The execution time of Earley's algorithm varies depending on the characteristics of the input grammar. It is thus likely that the modified Earley's algorithm will behave similarly. Richter [Ric85] states that there exist languages whose suffix languages will require $O(n^3)$ time to parse. However, experience with an implementation of the modified Earley's algorithm indicates that this is not the case for practical programming languages.

4.3. Partial Parses

The partial parse trees built by the modified Earley parser are important in the next chapter, and so will be discussed in greater detail here.

Initially, each newly created item represents the top of a (possibly empty) parse tree, and is marked as such. This mark is cleared only by the scanner and the completer.

Whenever a the scanner is applied to an item I_1 in state S_i and an item I_2 is added to state S_{i+1}, I_1 ceases to be the top of a parse tree, and I_2 becomes one. This corresponds the intuitive understanding of what the scanner is doing: since we have recognized yet another token of some rule, the new item that represents this becomes the one we are interested in. For example, if I_1=[$E \rightarrow E \cdot + T$,5] is the top of a parse tree and is processed by the scanner, then afterwards, I_1 is no longer the top of a parse tree, while the new item added to the successor state, I_2=[$E \rightarrow E + \cdot T$,5] is the top of some parse tree.

Assume that state S_i contains an item I_1=[$Z \rightarrow \beta \cdot, j$], and state S_j contains an item I_2=[$Y \rightarrow \alpha \cdot Z \beta, k$], where both I_1 and I_2 currently represent the tops of parse trees. When the completer is applied to I_1, a new item I_3=[$Y \rightarrow \alpha Z \cdot \beta, k$] is created and marked as the top of the parse tree. I_1 and I_2 cease to be tops of trees, since all information in them is now contained in I_3.

Consider the grammar for lists of identifiers given below:

$$G \rightarrow \vdash S \dashv$$

$$S \rightarrow x\ ,\ S$$

$$S \rightarrow x$$

We will follow examine the behaviour of the modified Earley parser on the invalid input string ω=$\vdash x, , x \dashv$. During parsing this string, the following sets of items are generated. States which are marked as the tops of parse trees are indicated with a superscript asterisk.

S_0: $[G \rightarrow \cdot \vdash S \dashv ; 0]$ Input: $\vdash$

S_1: $[G \rightarrow \vdash \cdot S \dashv ; 0]$ Input: x
$[S \rightarrow \cdot x, S ; 1]$
$[S \rightarrow \cdot x ; 1]$

S_2: $[S \rightarrow x \cdot, S ; 1]$ Input: ,
$[S \rightarrow x \cdot ; 1]$
$[G \rightarrow \vdash S \cdot \dashv ; 0]$

S_3: $[S \rightarrow x, \cdot S ; 1]^*$ Input: ,
$[S \rightarrow \cdot x, S ; 3]$
$[S \rightarrow \cdot x ; 3]$

After processing state S_3, state S_4 remains empty, which indicates that a syntax error has been seen. The token causing the error is used to generate the replacement for S_4, and processing continues with this state.

S_4: $[S \rightarrow (x) \cdot, S ; -]$ Input: ,

S_5: $[S \rightarrow (x), \cdot S ; -]$ Input: x
$[S \rightarrow \cdot x, S ; 5]$
$[S \rightarrow \cdot x ; 5]$

$S6$: $[S \rightarrow x \cdot, S ; 5]$ Input: $\dashv$
$[S \rightarrow x \cdot ; 5]$
$[S \rightarrow (x), S \cdot ; -]$
$[G \rightarrow (\vdash) S \cdot \dashv ; -]$

$S7$: $[G \rightarrow (\vdash) S \dashv \cdot ; -]^*$ Input: (none)

S_7 is the final state, since we have consumed the entire input. Two of the items above are marked as tops. The parse trees which correspond to them are show in Figure 4.1.

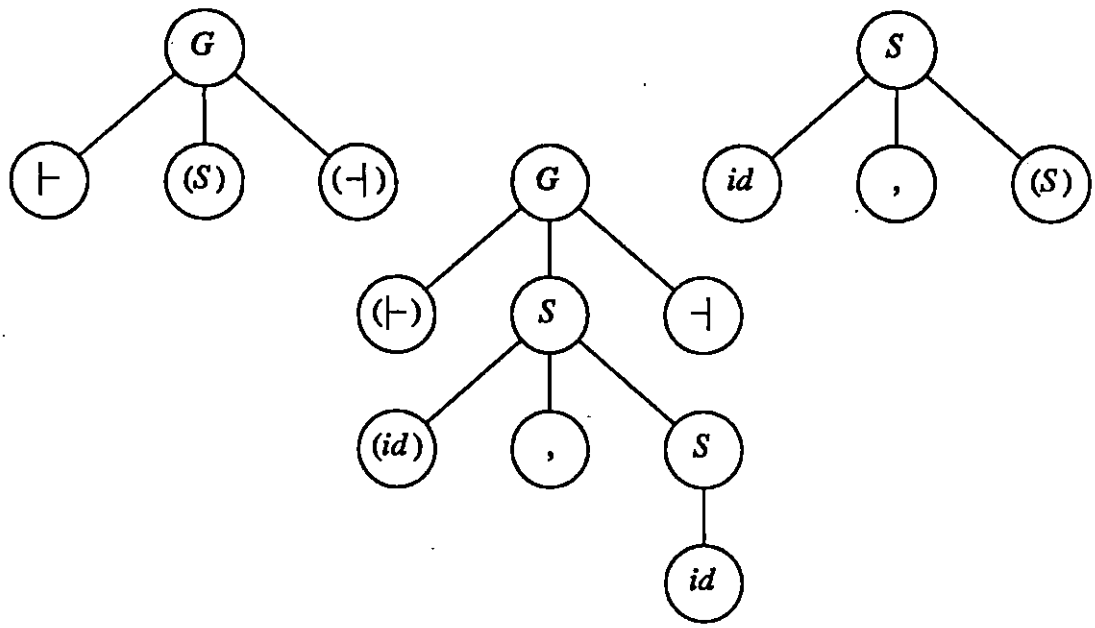

Figure 4.1. Partial Parse Trees *(symbols in parentheses are missing)*

4.4. Conclusions

In this chapter we have presented a form of Earley's algorithm for suffix parsing. In the next chapter, we describe an algorithm to find repairs for syntax errors based on information available in the states of the modified Earley parser.

CHAPTER 5

ERROR REPAIR

5.1. Introduction

The previous chapter described how the modified version of Earley's algorithm operates as a suffix parser, and generates a forest of partial parse trees. Once this portion of the algorithm is complete, the next step is to attempt to merge these partial parses into a single parse tree that represents a correct input string.

5.2. Error Repair

The modifications to Earley's algorithm described above allow the parser to detect errors independently of any attempt to repair them. The basic approach taken is to run the modified Earley algorithm described above on the input string, breaking it up into error-free substrings, and then to attempt to connect the substrings together.

Running the modified Earley algorithm on an input string effectively divides the string up into pieces $\omega_1, \ldots, \omega_k$. For $1 \le i \le k$, ω_i is a valid fragment, i.e., a non-empty substring of some string of the language recognized by the parser. Clearly, if $k=1$, then ω is a correct string. Otherwise, the boundary between ω_i and ω_{i+1} is the location of the i-th detected syntax error. We exclude the possibility of the first or last character being in error by augmenting the grammar for the language so that the goal production is of the form $\Sigma \rightarrow \vdash S \dashv$, and assuming that the symbols $\vdash$ and $\dashv$ are never in error. This ensures that both ω_1 and ω_k are at least one character long. For each valid fragment of the input string, there is a partial parse generated by the modified Earley parser. This partial parse is, in general, a forest of partial parse trees. The problem in error repair is how to reconnect these partial parses in the most natural way.

Consider the incorrect string $x+\!+x$ along with Grammar 4.1 from above. The partial parse trees generated by the modified Earley parser are shown in Figure 4.1. The subscripts on the terminal symbols indicate their position in the input stream.

We attempt to repair errors in the input in the order they were detected. Once the error point has been located, two scans are made in the vicinity of the error. One, moving backward in the input, searches for items representing uncompleted items. These items represent portions of the input that may have failed to complete before the syntax error. Note that items may be incomplete for other reasons as well, so there will be some extraneous information collected. We will refer to these incomplete items as *active* items. The other scan moves forward in the input, searching for completed suffix items. These suffix items represent the inferences we can make about what should have occurred prior to the error point. Our repair algorithm works by comparing these items pairwise, and attempting to find a pair that represents the "best" repair.

In comparing a suffix item and an active item, we attempt three different repairs on each pair. These operations are *identification* of one item with the other, *subordination* of the suffix item into the active item, or *subordination* of the the active item into the suffix item. Given two items, it may be possible to apply any, all, or none of these. These operations correspond to certain editing operation on parse trees.

Identification is essentially laying one item atop the other. For this operation to be valid, each item must represent the same production. For example, the active item

$$I_1 = [S \rightarrow x+\cdot(S), j]$$

and the suffix item

$$I_2 = [S \rightarrow (x)+S\cdot, -]$$

can be identified to give the item

$$[S \rightarrow x+S\cdot, j]$$

These items come from the parse of the string $x+\!+x$. Note that the two plus signs have become superimposed. This represents one possible repair — deleting a plus sign.

Subordination makes the parse tree represented by one of the items into a sub-tree of the other. For subordination to be possible, one of the items must have on the right-hand side of its production the symbol occurring on the left-hand side in the other item. Also, the corresponding tree motion must not move terminal symbols out of order with respect to the original input. In other words, a depth-first traversal of the leaves of the tree must still give the terminal symbols in the same order in which they occurred in the input. For example, I_1 could not be subordinated into I_2 since that would move an x from position one in the input past a comma from position three. We can, however, subordinate I_2 into I_1. The sub-parse tree shown in Figure 5.1 would result from this action. This repair corresponds to inserting an x between the two plus signs in $x+\!+x$.

5.3. The Algorithm

The actual algorithm for performing the search for repairs is relatively simple. It is shown in Figure 5.2. Once the work has been completed for this algorithm, we start

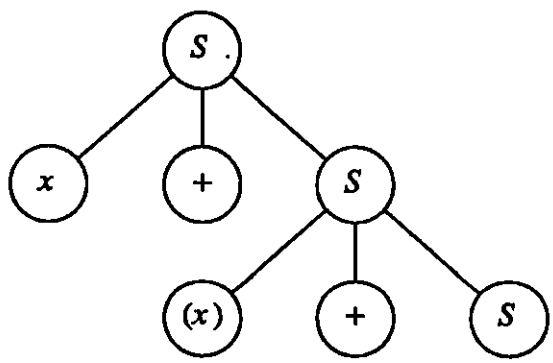

Figure 5.1. Parse Tree resulting from subordinating I_2 into I_1.

```
1.    scan for next error point
2.    collect active items between this error and the previous
3.    collect suffix items between this error and the following
4.    for each suffix item s do
4.1       for each active item a do
4.2           evaluate the repair using s and a
4.3           if s and a are the best repair so far
4.4               remember s and a
              fi
          od
      od
5.    generate an error message describing the repair based on a,s.
```

Figure 5.2. Repair Algorithm

again, looking for the next error. It is possible that either the set of active items, or the set of suffix items may be empty. In this case, as described here, we cannot repair this error. One possible modification to handle this situation would be to continue scanning past the next (or previous) error in the search for suffix (or active) items. This is not currently implemented.

Clearly, the interesting part of the algorithm is rating the various potential repairs in step 4.2, since this will determine exactly what repair is suggested.

The first step is to determine if any of the possible editing operations are applicable to items s and a. If s and a cannot be identified, and neither can be subordinated to the other, then no repair action is possible.

Next, for each possible editing operation, the repair is attempted. Each possible repair is essentially a complete *non-suffix* item that spans the error point. A new state is built, containing only this item, and the parser is started again with the input corresponding to the state in which s was found. New states are generated until (a) the next error or end of the input is reached, or (b) an error occurs that was not in the original parse. In case (a), the repair has been successful in fixing the error in question, and a numeric value is assigned to the repair based on the amount of change required to the original input text. In case (b), however, the repair is unsuccessful, and is discarded. If only one repair was successful in reaching the end of the error segment, then that represents the correct repair for the error. If more than one was successful, the one representing the least change in the input is used. If the error segments are very short, it may be the case that very many repairs are successful, because of insufficient context after the error to distinguish what was meant. In this case, although a repair will still be chosen, it is less likely to be a good repair.

This algorithm is outlined in Figure 5.3. Note that this algorithm could easily be run in parallel since the evaluation of each repair implied by each pair of active and suffix items can proceed independently. This is, however, a fairly coarse-grained parallelism.

5.4. Implementation

A simple version of the algorithm described above has been implemented, which checks only for repairs based on identification. The implementation has been tested on several different grammars, from one for strings of balanced parentheses to one for Pascal. In general, even this simplified version tends to choose the correct repair for most simple errors. Those cases in which the algorithm fails tend to be those where the error is at the beginning or end of a sequence. For example, when repairing the incorrect Pascal fragment

```
a := ( / b + c ),
```

the section `/ b + c` is replaced with the nonterminal *expression*, rather than inserting

```
1.    determine feasible editing operations
2.    for each feasible operation do
2.1       build a new state based on a and s
2.2       restart parsing from that state
2.3       if we successfully reach the end of this segment
2.4           this repair was successful
          fi
      od
3.    return the best repair
```

Figure 5.3. Evaluation Algorithm

an identifier before after the opening parenthesis.

5.5. Example

The output of the suffix parser on a small PL/0 program is shown in Figure 5.4. This program illustrates all of the different tyes of single-token errors — deletion of a single token, insertion of a single token, and mutation of a single token. The output of the error repair algorithm is shown in Figure 5.5. The two productions listed represent which items were identified to achieve the repair action indicated.

The final repair in Figure 5.5 requires a small amount of explanation. Any of +, −, *, or / could have been inserted to repair the syntax error. The only reason for the specific one chosen is that it was the first repair found by the repair algorithm that successfully repaired the error.

```
===============
Parsing file t8
| const a = 25
|       b = 36 ;
* -------^
*** Syntax Error on token ident, "b", near line 2 character 3
*** Expected one of:  ; ,
| var   b, c, , d;
* -------------^
*** Syntax Error on token ,, ",", near line 3 character 12
*** Expected one of:  ident
| begin
|       if b < c do a := 5;
* -----------------^
*** Syntax Error on token do, "do", near line 5 character 13
*** Expected one of:  then + - * /
|       a := a + b;
|       a := a 5
* --------------^
*** Syntax Error on token integer, "5", near line 7 character 10
*** Expected one of:  end ; . + - * /
| end.
|
```

Figure 5.4. Erroneous PL/0 Program

```
*** Attempting to repair errors ***

=-=-= attempting to fix error 1 =-=-=
<equate_list> ::= <equate_list> ### [,] [<one_equate>]
<equate_list> ::= [<equate_list>] [,] <one_equate> ###
Repair: insert ","

=-=-= attempting to fix error 2 =-=-=
<ident_list> ::= <ident_list> , ### [ident]
<ident_list> ::= [<ident_list>] , ident ###
Repair: delete ","

=-=-= attempting to fix error 3 =-=-=
<statement> ::= if <condition> ### [then] [<statement>]
<statement> ::= [if] [<condition>] [then] <statement> ###
Repair: replace "do" with "then"

=-=-= attempting to fix error 4 =-=-=
<term> ::= <term> ### [*] [<factor>]
<term> ::= [<term>] [*] <factor> ###
Repair: insert "*"
```

Figure 5.5. Output of Repair Algorithm

5.6. Conclusions

The repair algorithm described in this chapter makes use of the state information remaining after the suffix Earley parser has parsed the erroneous input string. The information contained in the active and suffix items is used to decide which repairs to attempt. These potential repairs are then attempted, and evaluated by restarting the parse at the appropriate point, and seeing which potential repair allows the most remaining input to be consumed.

A simplified version of the algorithm has been implemented, which has proven able to repair most common errors easily. In particular, most single token errors, i.e., insertion, deletion, and mutation errors are repaired correctly. Those that are not tend to occur at the beginning or end of bracketing structures, such as parentheses.

CHAPTER 6

CONCLUSIONS

6.1. Summary

In the preceding two chapters we have described a method of using Earley's algorithm as an error-correcting suffix parser. The method detects syntax errors independently of any particular repair. This is advantageous since it avoids altogether the problem of the "cascades" of error messages which can follow an incorrect repair. Once all errors detectable by the parser have been found, the error-repair algorithm is applied. Since text both before and after any particular error has been parsed, the error corrector has more information available to it to guide the repair procedure. This is similar in some aspects to the forward move algorithms discussed in Chapter 2. Even if error messages are limited to stating those symbols expected at each error point, this method can generate error reports at least as useful as those generated by many current compilers. This is due to the fact that the entire input is always scanned, and that cascades of errors are eliminated.

The modified Earley's algorithm and a simplified version of the error repair algorithm have been implemented. Experience indicates that the desired behaviour from a suffix parser is indeed attainable. In addition, simple repairs can be made, if so desired.

6.2. Future Research

Several areas remain to be explored.

The current repair algorithm assumes that errors are independent. That is, no error has any effect on neighbouring errors. To see that this is not always the case, consider the Pascal fragment

```
const a = 1;
      b = 2,
      c = 3;
```

Clearly, the error here is a mis-typed semicolon. However, when run through the Earley suffix parser, two errors are detected — one at the obvious place, the comma, and the other at the final semicolon. The second error occurs because the fragment

```
, b = 2
```

looks like part of either a list of parameters or a list of array subscripts, and in neither case can it be followed by a semicolon. This causes difficulty for the repair algorithm because there will be no suffix items in the error fragment

```
, b = 2
```

relating to constant declarations. Items which would allow a repair will occur only after the second error. One possible way of handling this would be to search successive error fragments for suffix items if the fragment immediately following the error did not contain sufficient information to lead to a repair. Alternatively, the token causing the error could be skipped before restarting the parse.

The basic algorithm could be extended to search for single-token repairs, i.e., insertion, deletion, or mutation of a single token at the point of the error, in addition to using the information gleaned from active and suffix items. The state of the Earley parser at the point of the error specifies a set of symbols which would have allowed some parsing action to occur. By using this information, and perhaps some heuristic knowledge as well, many single-token errors could be repaired without the time-consuming process of checking each pair of active and suffix items.

One interesting extension would be "reverse parsing." In this approach, the text around a syntax error is parsed both left-to-right, and right-to-left. In general, this will result in two different error points. The actual error *must* occur between these two points. For example, consider the Pascal fragment

```
if i = 1 do j := j + 1
```

On a forward (left-to-right) parse, the syntax error is detected at the `do` token. However, on a reverse parse, the error does not appear until the `if`. This indicates that the error must occur in the fragment `if i = 1 do`. More precisely, it indicates that this fragment cannot appear in *any* valid Pascal program.

The current implementation of the parser consists of approximately 2500 lines of C code. This implementation could be improved in several ways. In its current form, the parser dynamically builds and manipulates the Earley states. In addition to causing the program to consume large amounts of virtual memory, this also tends to make the execution rather slow. Making the suffix parsing algorithm table-driven would essentially eliminate both of these drawbacks, and should make the parsing speed similar to that for table-driven $LR(k)$ and $LL(k)$ parsers. Earley, in [Ear70], mentions, but does not describe, a table-driven version of his basic algorithm. The tables would probably be considerably larger than those for $LR(k)$ parsers, but table-compaction algorithms developed for $LR(k)$ tables could possibly be applied to keep the sizes of the tables in line.

REFERENCES.

[AhJ74] A. V. Aho and S. C. Johnson, LR Parsing, *Computing Surveys 6*, 2 (June 1974), 99-124.

[ASU86] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, Massachusetts, 1986.

[CoW73] R. W. W. Conway and T. R. Wilcox, Design and Implementation of a Diagnostic Compiler for PL/1, *Comm. ACM 16*, 3 (March 1973), 169-179.

[Cor87] G. V. Cormack, Deterministic Suffix Parsing, Unpublished manuscript, Department of Computer Science, University of Waterloo, June 1987.

[DeR71] F. L. DeRemer, Simple LR(k) Grammars, *Comm. ACM 14*, 7 (July 1971), 453-460.

[Ear70] J. Earley, An Efficient Context-Free Parsing Algorithm, *Comm. ACM 13*, 2 (February 1970), 94-102.

[GrR75] S. L. Graham and S. P. Rhodes, Practical Syntactic Error Recovery, *Comm. ACM 18*, 11 (November 1975), 639-649.

[GHJ79] S. L. Graham, C. B. Haley and W. N. Joy, Practical LR Error Recovery, *Conf. Rec. 6th ACM Symp. on Prin. of Programming Languages*, August 1979, 168-175.

[Iro63] E. T. Irons, An Error-Correcting Parse Algorithm, *Comm. ACM 6*, 11 (November 1963), 663-673.

[JaP73] E. B. James and D. P. Partridge, Adaptive Correction of Program Statements, *Comm. ACM 16*, 1 (January 1973), 27-37.

[JeW75] K. Jensen and N. Wirth, *Pascal User Manual and Report*, Springer-Verlag, 1975.

[Joh75] S. C. Johnson, Yacc — Yet Another Compiler-Compiler, Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey, July 1975.

[Len70] R. P. Lenius, *Error Detection and Recovery for Syntax Directed Compiler Systems*, Computer Science Department, University of Wisconsin, 1970.

[Lyo74] G. Lyon, Syntax-Directed Least-Errors Analysis for Context-Free Languages: A Practical Approach, *Comm. ACM 17*, 1 (January 1974), 3-14.

[MiM78] M. D. Mickunas and J. A. Modry, Automatic Error Recovery for LR Parsers, *Comm. ACM 21*, 6 (June 1978), 459-465.

[PaK79] A. B. Pai and R. B. Kieburtz, Global Context Recovery: A New Strategy for Parser Recovery from Syntax Errors, *Conf. Rec. 6th ACM Symp. on Prin. of Programming Languages*, August 1979, 158-167.

[Ric85] H. Richter, Noncorrecting Syntax Error Recovery, *ACM Trans. Prog. Lang. and Systems 7*, 3 (July 1985), 478-489.

[RiD78] G. D. Ripley and F. C. Druseikis, A statistical analysis of syntax errors., *Computer Languages 3*, (1978), 227-240.

[SiS] S. Sippu and E. Soisalon-Soininen, Practical Error Recovery in LR Parsers, *Conf. Rec. 9th ACM Symp. on Prin. of Programming Languages*, , 177-184.

VITA

Surname: HAGGLUND **Given Names:** RONALD DALE

Place of Birth: Ponoka, Alberta, Canada **Date of Birth:** August 19, 1964

Educational Institutions Attended, with Dates of Entering and Leaving:

University of Alberta, Edmonton 1982 to 1986
University of Victoria, B.C. 1986 to 1988

Degrees, Diplomas, Etc., Awarded, with Dates and Names of Institutions:

B.Sc. (Honours), University of Alberta, Edmonton 1986

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis (the title of which is shown below) to users of the University of Victoria Library, and to make *single copies only* for such users or in response to a request from the library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

A Modification to Earley's Algorithm for Syntax Error Recovery

Author 

January 25, 1988

Date