Static Semantics and Compiler Error Recovery

by

Robert Paul Corbett

June 1985

Sponsored by
Defense Advance Research Projects Agency (DoD)
Arpa Order No. 4871
Monitored by Naval Electronic Systems Command
under Contract No. N00039-84-C-0089

Static Semantics and Compiler Error Recovery

Robert Paul Corbett
Department of Electrical Engineering and Computer Science
Computer Science Division
University of California
Berkeley, California 94720

ABSTRACT

Good error recovery for compilers depends on accurate diagnosis of errors. When an error is misdiagnosed, the error message issued for it is apt to be misleading. Worse, the error recovery system may leave the compiler in a configuration that will cause spurious errors to be reported later. This dissertation presents new error recovery techniques for compilers that generally diagnose errors more accurately than earlier techniques.

The major innovation embodied in the new error recovery techniques is the use of general static semantic information to help detect and diagnose syntactic errors. There are usually many possible ways of recovering from an error. Testing if a potential recovery leads to semantic problems later involves executing the semantic actions associated with that recovery. If a potential recovery is rejected, the semantic actions that were performed while testing it must have no apparent effect on later compilation. Thus, it must be possible to undo the effects of semantic actions. For conventional compilers, the mechanisms needed to reverse the effects of semantic actions are too slow to be practical. A new compiler organization that permits semantic actions to be undone efficiently is presented. This new organization is suited for compiling languages, such as C, Pascal, and Ada, that require declarations to precede uses.

Two further ways of improving the performance of error recovery systems are considered. Error recovery systems sometimes fail to accurately diagnose an error because the parser has performed reductions based on the erroneous input. A variety of techniques for avoiding the adverse effects of such reductions are presented and compared. Also, a new panic mode algorithm for use with LR parsers is presented.

The new error recovery techniques have been applied in an error checking program for Pascal. The recoveries produced by that program are shown to compare favorably with those produced by two well known error recovery systems. Finally, some drawbacks of the new techniques and some directions for future work are discussed.

Acknowledgements

I would like to express my deep gratitude to my dissertation advisor, Professor Susan L. Graham, for her encouragement and understanding, and for her insights and assistance which contributed greatly to this work. I would also like to express my gratitude to the other members of my doctoral committee, Professors Paul N. Hilfinger and Robert M. Solovay.

I would like to offer special thanks to my fellow student Michael C. Shebanow, whose implementation of the algorithms presented in Chapter 7 was invaluable. I would like to thank Gerald Fisher and Michael Burke for freely sharing their ideas and their codes with me. I would like to thank Peter B. Kessler and Marshall K. McKusick for providing the graph profiler *gprof*.

I would like to thank Benjamin Zorn, Eduardo Pelegrí-Llopart, and Phillip Garrison for many discussions that helped clarify the concepts presented herein. I would like to thank all those who graciously offered their counsel, but especially Professors W. Kahan and Robert Fabry.

Particular thanks must be paid to my parents, Jeanette and Harvey Corbett, without whose support and encouragement this work would not have been possible.

Finally, I would like to offer thanks to the National Science Foundation, Grant MCS80-05144, and the Defense Advanced Research Projects Agency, Contracts N00039-82-C-0235 and N00039-84-C-0089, for their financial support over the years.

Table of Contents

Table of Figures

1

Introduction

Ideally, a compiler should detect and correctly identify every error in every program submitted to it. That goal, regrettably, is unattainable. Many errors either cannot be detected at compile time or are so difficult to detect that it is not practical to check for them. Even when an error is detected, it is, in general, impossible to correctly diagnose the error. Diagnosing an error involves guessing how a program deviates from the programmer's intent. Through heuristics, those guesses can be made highly accurate. Still, some failures must be expected.

The problem of providing good error recovery in a practical compiler is compounded by the need for efficiency. To handle errors well, a compiler must record information and perform tests that would otherwise be unnecessary. For example, to be able to associate locations with errors, the position of each symbol in the source text must be recorded. Those additional operations will cause the compiler to be slower. A slow compiler is as undesirable as one that does not handle errors well. A practical compiler must strike a balance between the efficiency and the power of its error recovery system.

Error recovery is a four step process. The four steps are detection, diagnosis, reporting, and patching. *Detection* consists of discovering the presence of an error. Errors are often classified according to the part of the compiler by which they are detected. Thus, errors detected by the lexical analyzer are called lexical errors, those detected by the parser are called syntax errors, and those detected by semantic action routines are called semantic errors. *Diagnosis* consists of guessing the location and nature of the error. The results of the diagnosis are used when reporting and patching the error. *Reporting* consists of providing the programmer with information to help him identify the error. *Patching* consists of modifying the state of the compiler so that compilation can continue.

Many error recovery techniques have been proposed. Among the most commonly used techniques are

1. **Error Productions.** If a compiler writer anticipates that certain syntax errors may occur, he can extend his grammar for the language to be compiled to include the erroneous constructs. Rules that are part of such extensions are called error productions. The compiler writer must provide for reporting errors handled by error productions.

2. **Local Recoveries.** A local recovery is a recovery that is determined by the immediate context in which the error was detected. Most local recovery algorithms consider only simple recovery actions such as insertion, deletion or replacement of single symbols. Local recovery algorithms usually do not require the compiler writer to supply any special information; any necessary information is inferred from the parser. Many algorithms allow the

compiler writer to supply a small amount of information that is used to fine-tune the choice of recoveries.

3. **Panic Mode.** A panic mode recovery consists of deleting symbols from the remaining input until a recognized symbol or sequence of symbols is at the head of the input. The parse stack is then reconfigured so that parsing can continue over the remaining input.

Good error recovery systems typically incorporate the three recovery techniques mentioned above and perhaps others as well. Recoveries that involve changes to the program text at the token level only are customarily called *repairs*.

In theory, all syntax errors could be handled by error productions. If the grammar for a language is extended to accept all possible input strings, no other syntactic error recovery capabilities need be provided. However, a grammar capable of distinguishing erroneous syntax from legal syntax for any input is apt to be large and of a form for which efficient parsers cannot be constructed. Therefore, error productions are normally used only for errors that cannot be handled well using other recovery techniques.

Error productions are often used to relax restrictions in the language to be compiled. For example, in Pascal [ANS83], declarations must appear in a fixed order. If a declaration occurs out of order, there is little chance that the error could be patched by a local recovery. A panic mode recovery for such an error would be tantamount to deleting the declaration. Many spurious semantic errors result from such a recovery. Therefore, extending the grammar to allow declarations to appear in any order appears to be the only way to handle such errors gracefully.

Local recoveries work best for simple errors. For example, consider the erroneous Pascal code fragment

$$\begin{array}{l} i := i + 1 \\ j := 0; \end{array}$$

where i and j are integer variables. The likely error is that a semicolon has been omitted from the end of the first line. A good repair algorithm should determine that a semicolon should be inserted between the two lines. A statistical study of errors in Pascal programs [RD78] has shown that, for Pascal at least, local recovery techniques should be effective for most common errors.

There are often many different local recoveries that could be used to patch an error. For example, suppose the erroneous statement

$$a := m);$$

appears in a Pascal program. The apparent error is that the statement contains an unmatched right parenthesis: The error could be patched by inserting a left parenthesis before the identifier m or by deleting the right parenthesis. Either repair would seem reasonable. However, the error could be patched just as effectively by replacing the right parenthesis with a semicolon. That repair is apt to seem unreasonable to most programmers. Many local recovery algorithms allow a compiler writer to bias the choice of recoveries in favor of those he feels are desirable. The compiler writer is allowed to assign costs to each possible recovery. Whenever there is a choice of local recoveries that patch an error, the recovery whose cost is the lowest is selected.

Panic mode recoveries are useful when an error deviates so far from a legal text that no simple correction can patch the error. Suppose, for example, that the Algol-like statement

for i := 1 **step** 1 **until** n **do** S

appears in a Pascal program. It is unlikely that most local recovery algorithms could patch such an error. In such cases, a panic mode recovery may provide the only way to patch the error. Nonetheless, panic mode recoveries should only be used as a last resort since the error messages produced by panic mode algorithms are usually less helpful than those produced by other error recovery techniques.

The main goal of this work has been to develop techniques for diagnosing errors more accurately than is done by earlier error recovery systems. Accurate diagnosis is the key to good error recovery. The diagnosis of an error largely determines the way in which the compiler will recover from it. If a diagnosis is incorrect, the resulting recovery may cause spurious errors to be detected. Most good error recovery algorithms diagnose an error by testing several possible diagnoses and selecting the one that seems the best. One way of improving that process is to increase the number of diagnoses that are considered since the correct diagnosis cannot be chosen if it is not considered. Another way is to improve the criteria for selecting recoveries. Both techniques have been applied in this work.

Most existing error handlers make no use of semantic information when diagnosing syntax errors. As a result, they sometimes choose diagnoses that lead to spurious semantic errors later. Consider the erroneous Pascal statement

if i j **then** *skip*

where i and j are integer variables. The probable error in this instance is that a relational operator has been omitted between i and j. This example was submitted to two well-known error handling systems. One of them recovered from the error by inserting the operator '+' between i and j; the other by deleting j. In both cases, the expression created to replace the predicate expression of the if-statement is of type integer. Thus, both recoveries cause a spurious semantic error to be detected later since the type of a predicate expression must be Boolean.

The major innovation of this work has been the creation of techniques for using general static semantic information to help detect and diagnose syntax errors. Error recovery algorithms that take advantage of semantic information will be called *semantics-directed*. As the previous example demonstrates, use of semantic information can improve the choice of error diagnoses. If semantic information is to be used to help diagnose syntax errors, it must be available while parsing. Hence, semantic analysis must be carried out in tandem with parsing. Further, any semantic actions performed while testing a potential diagnosis must not affect later stages of compilation if that diagnosis is rejected. Therefore, semantics-directed error recovery requires that the compiler be organized so that the effects of semantic actions can be reversed.

Syntax errors often go undetected until after the parser has performed actions that make it hard to recover from them. For example, bottom-up parsers sometimes perform reductions without examining the token to the right of the symbols involved in the reductions. Consider, for example, the erroneous Pascal statement

n := m, + 1

The apparent error is the presence of a comma in a context where commas are not allowed. If the error is detected before any reductions involving m are performed, a good local recovery algorithm should determine that the comma should be deleted. However, some parsers for Pascal will reduce the text preceding the comma to a statement in spite of the presence of the comma. Unless the effects of the erroneous reductions can be reversed, it is unlikely a good recovery will be found. An analogous problem for top-down parsers is discussed by Burke and Fisher [BF82].

Many good panic mode algorithms have been developed for top-down parsers. The panic mode algorithms that have been proposed for LR parsers do not work nearly so well. As a part of this work, an improved panic mode algorithm for LR parsers has been developed.

An implementation is the best test of an error handling system. Many impractical error handling techniques have been described in the literature. With but few exceptions, those techniques either have not been implemented or have been implemented only for unrealistic languages that do not expose their flaws. The new error recovery techniques described in later chapters have been implemented as part of an error checking program for Pascal called the Pascal auditor. Measurements of the Pascal auditor's speed and space requirements show the practicality of the new techniques.

A new parser generator named Bison has been written to assist construction of compilers using the new error recovery techniques. Bison was designed to support experiments with a variety of error recovery techniques. The parsers produced by Bison are faster than those produced by most other parser generators. Furthermore, Bison itself is faster than most other parse generators because it is based on more modern algorithms.

As a demonstration of the power of the new error recovery techniques, the Pascal auditor has been compared with two well-known error handling systems. Ripley and Druseikis [RD78] have created a sample of erroneous Pascal programs that has become a standard test suite for error handling systems. The recoveries produced by the Pascal auditor for that test suite have been compared with those produced by the other systems.

The remaining chapters are organized as follows. The next chapter introduces the terminology and notation used in later chapters. Chapters 3 through 5 describe schemes for using semantics to help detect and recover from errors. Chapter 6 explores techniques for preventing or reversing the effects of erroneous reductions. Chapter 7 presents the new panic mode algorithm. Chapter 8 describes the Pascal auditor and the empirical data obtained from it. The final chapters discuss lessons learned from the implementation, directions for future work, and conclusions. All examples of errors presented in the remaining chapters are taken from Pascal programs unless stated otherwise.

2
Terminology

Let S be a set of symbols. A ***string*** over S is a finite sequence of symbols in S. The empty sequence is called the ***empty string*** and is denoted by the Greek letter λ. The ***length*** of a string $a_1 \ldots a_n$ is n. For any symbol a, a^k is the string consisting of k instances of a. S^k is the set of all strings over S of length k. S^* is the set of all strings over S (including λ). Let $x = a_1 \ldots a_m$ and $y = b_1 \ldots b_n$ be any two strings. The ***concatenation*** of x and y (in that order) is the string $a_1 \ldots a_m b_1 \ldots b_n$. Concatenation is indicated by adjacency. For example, the concatenation of x, y, and z is denoted as xyz. A string x is a ***prefix*** of a string y if and only if $y = xz$ for some $z \in S^*$.

Let V be a finite set of symbols, and let Σ be a proper subset of V. Let N denote $V - \Sigma$. A ***production*** or ***rule*** over V and Σ is an ordered pair (A, x) where $A \in N$ and $x \in V^*$. A production (A, x) is denoted as $A \rightarrow x$. For any rule $A \rightarrow x$, A is its ***left-hand side*** (lhs) and x is its ***right-hand side*** (rhs).

A ***context-free grammar*** G is a 4-tuple (V, Σ, P, S), where V is a finite set of symbols, Σ is a proper subset of V, P is a finite set of productions over V and Σ, and $S \in V$. N denotes the set $V - \Sigma$. A symbol in Σ is a ***terminal symbol***, and a symbol in N is a ***nonterminal symbol***. The symbol S is the ***start symbol***.

For any two strings $x, y \in V^*$, the relation $x \Rightarrow y$ is true if and only if $x = sZt$, $y = szt$, and $Z \rightarrow z \in P$, for some $Z \in N$ and $s, t, z \in V^*$. The relation $x \underset{rm}{\Rightarrow} y$ is true if and only if $x = sZt$, $y = szt$, and $Z \rightarrow z \in P$, for some $Z \in N$, $t \in \Sigma^*$, and $s, z \in V^*$. The symbol $\overset{*}{\Rightarrow}$ denotes the reflexive transitive closure of $\Rightarrow$, and $\underset{rm}{\overset{*}{\Rightarrow}}$ denotes the reflexive transitive closure of $\underset{rm}{\Rightarrow}$. A string x ***derives*** a string y if and only if $x \overset{*}{\Rightarrow} y$.

A string $x \in V^*$ is a ***sentential form*** of G if and only if $S \overset{*}{\Rightarrow} x$. A ***sentence*** of G is a sentential form x such that $x \in \Sigma^*$. The ***language*** defined by G is the set of all sentences of G and is denoted as $L(G)$. A string x is a ***correct prefix*** if and only if x is the prefix of a sentential form. A string x is a ***right sentential form*** if and only if $S \underset{rm}{\overset{*}{\Rightarrow}} x$. Let $x = szt$ be a string such that $S \underset{rm}{\overset{*}{\Rightarrow}} sZt \underset{rm}{\Rightarrow} szt$. Then z is a ***handle*** of x.

A ***derivation tree*** T of G is a labeled ordered tree such that

1. Each interior node is labeled with a nonterminal symbol.
2. Each leaf node is labeled with a terminal symbol or λ.
3. For each interior node ν, let $\nu_1, \ldots, \nu_n$ be the immediate descendants of ν. Let A be the symbol labeling ν. Then either
   a) $n = 1$, ν_1 is labeled with λ, and $A \rightarrow \lambda \in P$, or
   b) $\nu_1, \ldots, \nu_n$ are labeled with the symbols $a_1, \ldots, a_n$ respectively, and $A \rightarrow a_1 \ldots a_n \in P$.

A ***parse tree*** is a derivation tree whose root node is labeled with the start symbol S. The ***frontier*** of a derivation tree T is the string formed by concatenating the symbols labeling the leaves of T in left to right order. The frontier of every parse tree is a sentence of G.

Every sentence is the frontier of at least one parse tree.

An LR(k) *parsing automaton* M is an 8-tuple $(Q, V, \Sigma, P, f, g, q_0, \$)$ where Q is a finite set of states, V is a finite set of symbols, Σ is a proper subset of V, P is a finite set of productions over V and Σ, f and g are functions, $q_0 \in Q$, and $\$$ is a symbol such that $\$ \notin V$. Q is the *state set* of M. N denotes $V - \Sigma$. The symbols in Σ are the *terminal symbols* and the symbols in N are the *nonterminal symbols*. $V_\$$ denotes $V \bigcup \{\$\}$, and $\Sigma_\$$ denotes $\Sigma \bigcup \{\$\}$. The *action set AS* of M is the set

$$AS = \{\textbf{accept}, \textbf{shift}, \textbf{error}\} \bigcup \{\textbf{reduce}\, p \mid p \in P\}.$$

The function f is the *action function* of M; it is a total function of the form $f: Q \times \Sigma_\$^k \rightarrow AS$. The function g is the *goto function* of M; it is a partial function of the form $g: Q \times V \rightarrow (Q - \{q_0\})$. For every state $q \in Q$ other than q_0, there is exactly one symbol a such that $g(p, a) = q$ for some $p \in Q$. The symbol a is called the *accessing symbol* of q. The state q_0 is the *start state*. The symbol $\$$ is the *endmarker*.

A *configuration* C of an LR(k) parsing automaton M is an ordered pair (Γ, x) where Γ is a finite nonempty sequence of states in Q, and $x \in \Sigma_\$^*$. The string x must be of the form $y\$^k$ where $y \in \Sigma^*$. For any string $x \in \Sigma^*$, the *initial configuration* of M for x is $(q_0, x\$^k)$.

A *move* of an LR(k) parsing automaton M is a transition from one configuration to another. For each configuration C, there is at most one configuration C' such that the transition from C to C' is a move. The relation between C and C' is denoted as $C \vdash C'$. The relation $\vdash$ is determined by the action and goto functions of M. Let $C = (\Gamma, x)$ where $\Gamma = q_{i_1}, \ldots, q_{i_m}$ and $x = a_1 \ldots a_n$. Then the *action* α determined by M for C is $\alpha = f(q_{i_m}, a_1 \ldots a_k)$. If $\alpha = \textbf{shift}$, then $C \vdash (q_{i_1} \ldots q_{i_m} q, a_2 \ldots a_n)$ where $q = g(q, a_1)$. If $\alpha = \textbf{reduce}\, p$ where $p = A \rightarrow b_1 \ldots b_t$, then $C \vdash (q_{i_1} \ldots q_{1_{m-t}} q, x)$ where $q = g(q_{m-t}, A)$. If $\alpha = \textbf{accept}$ or $\alpha = \textbf{error}$, there is no move from C.

A configuration $C = (q_{i_1} \ldots q_{i_m}, x)$ is an *accepting configuration* if and only if $x = \$^k$ and $f(q_{i_m}, \$^k) = \textbf{accept}$. The parsing automaton M *accepts* a string x if and only if there is a sequence of configurations $C_1, \ldots, C_n$ such that C_1 is the initial configuration of M for x, C_n is an accepting configuration, and $C_1 \vdash \cdots \vdash C_n$. The *language* of M is the set of all strings that M accepts and is denoted as $L(M)$.

A configuration $C = (q_{i_1} \ldots q_{i_m}, a_1 \ldots a_n)$ is an *error configuration* if and only if $f(q_{i_m}, a_1 \ldots a_k) = \textbf{error}$. When the parsing automaton enters an error configuration, it *detects* an error.

An *attribute grammar AG* consists of a context-free grammar $G = (V, \Sigma, P, S)$ augmented with attributes, semantic functions, and dependency vectors. For each symbol $X \in V$, there is a finite set $A(X)$ of attributes. $A(X)$ is partitioned into two disjoint subsets, the *inherited attributes* $I(X)$ and the *synthesized attributes* $S(X)$. The set $I(S)$, where S is the start symbol, must be empty. If X is a terminal symbol, then $S(X)$ must be empty. The union of $A(X)$ for all $X \in V$ is A. Each attribute $\alpha \in A$ is associated with a (possibly infinite) universe U_α of values.

For each production $p = X_0 \rightarrow X_1 \ldots X_{n_p} \in P$, there is an associated set of semantic functions and dependency vectors. For each synthesized attribute σ of X_0, there is a function $f^p_{0\sigma}$. For each inherited attribute ι of X_k, where $1 \leq k \leq n_p$, there is a function $f^p_{k\iota}$. Each function $f^p_{k\alpha}$ is defined over

$$U_{\alpha_1} \times \cdots \times U_{\alpha_m} \rightarrow U_\alpha,$$

where $m \geq 0$, and $\alpha_i \in A$ for $1 \leq i \leq m$. The arity of each semantic function may be different.

Each semantic function $f^p_{k\alpha}$ is paired with a dependency vector $d^p_{k\alpha}$. A dependency vector indicates which values of the attributes of the symbols $X_0,\ldots,X_{n_p}$ are to be the arguments of the matching semantic function. The number of elements in each dependency vector must equal the arity of the corresponding semantic function. An attribute α of the symbol X_i, $1 \leq i \leq n_p$, can be represented by the ordered pair (i, α). Each element of a dependency vector is an ordered pair of that form. If the j-th element of $d^p_{k\alpha}$ is (i, α), then the domain of the i-th argument of $f^p_{k\alpha}$ must be U_α. The *dependency set* $D^p_{k\alpha}$ is the union of the elements of $d^p_{k\alpha}$.

Let *AG* be an attribute grammar, and let G be its underlying context-free grammar. An *attributed parse tree APT* of *AG* is a parse tree T of G together with a function μ. The function μ is the *meaning* of the tree. The domain of μ is the set

$$S = \{\ (\nu, \alpha) \mid \nu \textit{ is a node of } T, \textit{ and } \alpha \in A(X) \textit{ where } X \textit{ is the symbol labeling } \nu\ \}.$$

For $(\nu, \alpha) \in S$, $\mu(\nu, \alpha) \in U_\alpha$. If $(\nu, \alpha) \in S$, then $\mu(\nu, \alpha)$ is the *value* of α at ν. An *APT* is an *evaluation* of the parse tree T if an only if

1. T is the underlying parse tree of the *APT*.
2. For each interior node ν of T whose sole descendant is labeled with λ, let X be the symbol labeling ν and let $p = X \rightarrow \lambda$. For each $\sigma \in S(X)$, $\mu(\nu, \sigma)$ must equal $f^p_{0\sigma}(\mu(\nu, \alpha_1),\ldots,\mu(\nu, \alpha_m))$, where m is the arity of $f^p_{0\sigma}$, and $d^p_{0\sigma} = ((0, \alpha_1),\ldots,(0, \alpha_m))$.
3. For each interior node ν of T whose immediate descendants $\nu_1,\ldots,\nu_n$ are labeled with symbols in V, let $\nu_0 = \nu$, and let $X_0,\ldots,X_n$ be the symbols labeling $\nu_0,\ldots,\nu_n$ respectively. Let $p = X_0 \rightarrow X_1\ldots X_n$. For each attribute $\sigma \in S(X_0)$, $\mu(\nu, \alpha)$ must equal $f^p_{0\sigma}(\mu(\nu_{i_1}, \alpha_1),\ldots,\mu(\nu_{i_m}, \alpha_m))$, where m is the arity of $f^p_{0\sigma}$, and $d^p_{0\sigma} = ((i_1, \alpha_1),\ldots,(i_m, \alpha_m))$. Similarly, for $1 \leq k \leq n$ and for each inherited attribute $\iota \in I(X_k)$, $\mu(\nu_k, \iota)$ must equal $f^p_{k\iota}(\mu(\nu_{i_1}, \alpha_1),\ldots,\mu(\nu_{i_m}, \alpha_m))$, where m is the arity of $f^p_{k\iota}$, and $d^p_{k\iota} = ((i_1, \alpha_1),\ldots,(i_m, \alpha_m))$.

In other words, an *APT* is an evaluation if and only if the values assigned to the attributes are consistent with the values of the semantic functions for those attributes.

Let $p = X_0 \rightarrow X_1\ldots X_n$. Let α be an attribute of X_k, where $1 \leq k \leq n$. The *local closure* $\overline{D}^p_{k\alpha}$ of $D^p_{k\alpha}$ is the smallest set such that

1. $D^p_{k\alpha} \subset \overline{D}^p_{k\alpha}$, and
2. if $(i, \alpha') \in \overline{D}^p_{k\alpha}$, then $D^p_{i\alpha'} \subset \overline{D}^p_{k\alpha}$.

An *L-attributed grammar* is an attribute grammar *AG* such that for every rule $p = X_0 \rightarrow X_1\ldots X_n$ of the underlying context-free grammar of *AG*

1. if $\sigma, \sigma' \in S(X_0)$, then if $(0, \sigma') \in \overline{D}^p_{0\sigma}$, $(0, \sigma) \notin \overline{D}k\sigma'^p$,

2. if $\iota \in I(X_k)$, $1 \le k \le n$, then for all $(i, \alpha) \in \overline{D}^p_{k\iota}$, $i \le k$, and
3. if $\iota \in I(X_k)$, $1 \le k \le n$, then for all $(k, \alpha) \in \overline{D}k\iota^p$, $\alpha \in I(X_k)$ and $(k, \iota) \notin \overline{D}^p_{k\alpha}$.

These restrictions ensure that it is possible to evaluate the attributes of any parse tree in a single top-down left-to-right pass over that tree.

3

Previous Proposals for Semantics-directed Error Recovery

The idea of using semantics to help detect and recover from syntactic errors is not new. Many papers on syntactic error recovery suggest possible uses for semantic data. Most of them, however, do little more than mention that those possibilities exist. Still, some substantial work has been done in this area. At least two existing compilers use some static semantic data to assist in error recovery.

Graham and Rhodes [GR75] were among the first to suggest that semantic information could aid in syntactic error recovery. At the end of their paper, they speculate on ways to improve their error recovery system. As an example of the possible uses of semantics, they suggest that when the recovery algorithm inserts an identifier, semantics might be used to decide which identifier should be inserted. That example seems ill-chosen. While semantic information might preclude a particular identifier from appearing in a given context, it rarely determines that a particular identifier must appear in that context. Graham and Rhodes also outline a scheme for permitting semantic analysis to continue after recovering from a syntax error.

Other papers on syntactic error recovery also mention possible uses for semantic data. Pennello and DeRemer [PD77] consider ways of permitting semantic analysis to continue following recoveries from syntax errors. Mickunas and Modry [MM78] come closer to the ideas developed in this work. They suggest that semantic information might be usefully employed in choosing a recovery. They do not, however, suggest how to do so.

Milton, Kirchhoff, and Rowland [MKR79] made a serious attempt to include semantics-directed error recovery as part of an attribute grammar based compiler generator. Their compiler generator is unusual in that the parsers it produces can use semantic information to help decide which parsing actions to perform. Their error recovery algorithm is derived from the algorithm proposed by Fischer, Milton, and Quiring [FMQ80]. The algorithm uses tables defined over symbols and attribute values. They state that the implemented algorithm does not use semantics because the tables would become too large. In saying so, they understate the problem. For any real programming language, the set of attribute values will be infinite. Therefore, the tables needed by the error recovery algorithm will also be infinite. Presumably, they would avoid this problem by restricting the set of attribute values that index the tables to some finite range.

In her dissertation [Sch82], Cosima Schmauch presents a more practical proposal for semantics-directed error recovery. She too advocates using attribute grammars to define the static semantics of programming languages. However, in her scheme, the semantic checks associated with each rule are separated from the semantic functions that define the attributes' values. Those semantic checks are called *primitive predicates*. She proposes using the primitive predicates to guide the choice of recoveries. A recovery from a syntax error will be favored if it does not cause a primitive predicate to be violated. If every recovery causes a primitive predicate to be violated, the recovery that permits the greatest number of reductions before such a violation occurs is chosen. Because her algorithm was not implemented, it is hard to judge how well it might work. Intuitively, it seems that her scheme should produce good recoveries, but that compilers using it will be too slow to be practical.

The error recovery algorithm of Feyock and Lazarus [FL76] makes significant use of semantic information. Their system assumes that the symbol table will be created in tandem with parsing. A global variable is used as a semantic error flag. Whenever a reduction is done according to a rule that contains an identifier or constant on the right-hand side, a check is made to determine if the semantics of that symbol are correct for the given context. If a semantic error is detected, the semantic error flag is set to true. For example, the semantic error flag is set to true if an undeclared identifier appears on the left-hand side of an assignment. When testing a possible recovery, the error recovery algorithm initially sets the semantic error flag to false. The test consists of modifying the parse stack according to the potential recovery and then parsing the following input text. Semantic checks are made during the test. The recovery is rejected if a syntactic error was discovered during the test or if the semantic error flag was set to true.

Feyock and Lazarus used their system in a compiler for XPL [MHW70] called EXPL. They found that EXPL spent between 2 to 3 seconds on average for each error detected when running on an IBM 360/50. Although they state that that speed is acceptable, it is at least a dozen times slower than other error recovery algorithms that produce comparable results. Their use of semantics is not the major cause of the algorithm's inefficiency; rather, it is the algorithm's syntactic component that causes it to be slow. The cost of checking for semantic errors, though significant, is comparatively small.

The paper by Feyock and Lazarus lacks detail. Many important questions about their system can only be answered by inference from the examples given at the end of the paper. Apparently, no semantic checks are made while parsing declarations, and no semantic actions are performed while testing potential recoveries. Those restrictions contribute to the efficiency and simplicity of their method but limit its power.

The sample results presented by Feyock and Lazarus are very good. However, the examples appear to have been selected to show their system at its best. If their system works as described in their paper, it cannot handle multiple errors. If a statement contains two or more independent errors, all recoveries from those errors will be rejected. Feyock and Lazarus did not give any examples of statements that contained multiple errors in their paper. Also, they admit that if their error recovery algorithm finds more than one viable recovery, it does not do a good job of selecting among them. However, none of their examples illustrate this problem.

The Feyock and Lazarus scheme for applying semantics to error recovery suffers from serious limitations. Their system apparently does not perform semantic actions while testing potential recoveries. Therefore, if the semantic action associated with a rule used in a reduction would normally alter the contents of the symbol table, the symbol table will be left unchanged while testing a potential correction. That limitation does not appear to be a problem for their XPL compiler. For other languages, however, it could prove a serious deficiency. For example, in Pascal, if a syntax error were detected in a with-statement and the semantics actions associated with that statement were not executed while testing potential recoveries, good recoveries might be rejected because of spurious semantic errors. Another limitation of their system is that it does not perform general semantic checks while testing potential recoveries. The only checks they do are to check that identifiers and constants appear in syntactic contexts where they are semantically legal. That deficiency may prove a serious liability for languages with complex semantics, such as Ada† [DoD83].

The error recovery algorithm by Graham, Haley, and Joy [GHJ79] is the best known recovery algorithm that incorporates semantics. The Graham-Haley-Joy algorithm is implemented as a part of the Berkeley Pascal compiler and interpreter. It is similar to the Feyock-Lazarus algorithm in that whenever an identifier is encountered in a context where a particular class of identifier is required, a check is made to see if the identifier is of that class. The methods differ in their responses to errors detected by those checks. Instead of setting a flag to signal that an error has been discovered, the Graham-Haley-Joy system invokes the error handler. The error recovery algorithm first assigns a tentative cost to changing the identifier's semantics to the desired semantics. A test is then done to see if parsing can continue after that change. If a new error is found during the test, the cost associated with changing the semantics is increased. After the cost of the semantic change has been computed, a number of syntactic changes are also tested. The potential recovery assigned the lowest cost is selected as the recovery to be applied.

One reason for the difference between the ways Berkeley Pascal and EXPL treat semantic errors is that Berkeley Pascal uses a less powerful but more efficient syntactic error recovery scheme. The error recovery algorithm used by EXPL is able to back up the state of the parse. Therefore, delaying detection of an error does not preclude finding the best recovery. However, the execution time costs associated with providing the ability to back up the parse are large. Partly for that reason, Berkeley Pascal does not include the ability to back up the parse. Therefore, if it delayed detecting semantic errors, it would sometimes be unable to find good recoveries. Consider the statement

$$a(i] := 0$$

where a is an array variable. The likely error is that a left parenthesis has been used where a left square bracket was intended. When a is encountered, the semantic check is made to see if a is the name of a procedure. The check fails, and so the error recovery algorithm is invoked. Eventually, the error recovery system replaces the left parenthesis with a left square bracket, and normal parsing resumes. Had the semantic check had not triggered the error recovery algorithm, the parser would treat the text up to the right square bracket as the start of a procedure statement. The error recovery algorithm would by then be unable to find a good recovery.

Berkeley Pascal's error recovery algorithm does not always make good use of the semantic information available to it. Consider, for example, the erroneous Pascal statement

$$p[x + 1)$$

where p is declared to be a procedure of one parameter. The error is that a left square bracket has been used where a left parenthesis was intended. The left square bracket causes Berkeley Pascal to test if p is an array variable. Since it is not, the error recovery algorithm is invoked. The error recovery algorithm decides that the best recovery is to treat p as an array identifier. Then, when the parser reaches the right parenthesis, the

† Ada is a registered trademark of the U. S. Government — Ada Joint Project Office.

error recovery algorithm is reinvoked. This time, it is unable to find a good local recovery. Therefore, it resorts to panic mode and eventually reports that a malformed expression has been found. The poor choice of a recovery illustrated by this example would have been avoided by increasing the cost of replacing an identifier with an identifier of another class.

A major limitation of the schemes for applying semantics to syntactic error recovery proposed by Feyock and Lazarus and by Graham, Haley, and Joy is that they cannot take advantage of all of the semantic information that is available at compile time. The only semantic data they use is information about identifiers obtained from the symbol table and information about constants obtained from the lexical analyzer. Semantic data generated during semantic analysis is unavailable to either system because they both delay semantic analysis until after parsing has been completed.

4

Semantics-directed Error Recovery

This chapter considers ways of extending existing local recovery algorithms to take advantage of semantic data. It begins with a survey of local recovery algorithms for LR parsers. The survey is followed by a discussion of techniques for using semantic data to enhance those algorithms. Finally, the question of how semantic data should be supplied to the error recovery routines is examined.

4.1 Local Recovery Algorithms for LR Parsers

Some local recovery algorithms can only be used with specific classes of parsers. This section surveys local recovery algorithms that can be used with LR parsers. LR parsing and its related subclasses are probably the most widely used table-driven parsing techniques. Therefore, local recovery algorithms that do not work for LR parsers are of lesser interest.

A parser is a *correct prefix parser* if and only if for every input string $a_1 \dots a_k a_{k+1} \dots a_n$ such that $a_1 \dots a_k$ is a correct prefix but $a_1 \dots a_k a_{k+1}$ is not, the parser will not advance over a_{k+1} before detecting an error. A class of parsers possesses the *correct prefix property* if and only if every parser in that class is a correct prefix parser. Every LR parser is a correct prefix parser.

Let $x = a_1 \dots a_k a_{k+1} \dots a_n$. Suppose that a parser has detected an error after advancing over a_k but before advancing over a_{k+1}. Then the position in x between a_k and a_{k+1} is the error's *detection point*. An error's detection point need not be its true location. Consider, for example, the code fragment

$$\begin{array}{l} i := i + 1; \\ a[i - j + 1] = 0 \ \textbf{then}\ S; \end{array}$$

The likely error here is that the keyword **if** has been omitted from the start of the second line. However, a normal LR(1) parser will not detect an error until it has shifted over the right bracket. Therefore, the error's detection point will be between the right bracket and the equals sign.

Lévy [Lév75] was among the first to propose a local recovery algorithm for LR parsers. Lévy's algorithm is related to the *minimum distance* repair algorithms [AP72, Lyo74]. The only types of recoveries Lévy considers are insertion and deletion of tokens. There is an *a priori* bound N on the number of insertions and deletions permitted for a single error.

Let $x = a_1 \dots a_k a_{k+1} \dots a_n$ be a terminal string. Suppose an error has been detected in x and that its detection point is between a_k and a_{k+1}. Lévy's algorithm begins by performing a *backward move*. The input text is examined to determine the leftmost position $i \leq k$ in x such that deleting a_i or inserting a token immediately to the left of a_i could be part of a recovery. The parser's configuration is then backed up to the configuration it had just after shifting over a_{i-1}; if $i=1$, the parser is restored to its initial configuration. Consider, for example, the program shown in Figure 4.1.

```
program max(input, output);
    var x, y: real;
begin
    readln(x);
    readln(y);
    if x := y then
        writeln(x)
    else
        writeln(y)
end.
```

Figure 4.1 Example illustrating backtracking in Lévy's algorithm

The error is that the token ':=' appears in the if-statement's predicate expression. It seems likely that the programmer meant to write '>=' but made a typographical error. Assume $N = 1$. Then the leftmost token in the program that could be involved in a recovery is the keyword **if**. Therefore, the parser will be backed up to the configuration it had just after shifting over the semicolon preceding the keyword **if**.

After completing the backward move, the algorithm performs a *forward move*. Every modification of the input starting from position i that involves at most N insertions or deletions is considered. The forward move consists of a parallel parse over all of the modified strings. When the parse over one of the modified strings enters an error configuration, that string is dropped from consideration. The forward move continues until either all of the possible repairs are eliminated or all of the repairs still under consideration enter equivalent configurations. In the first case, presumably, the error recovery algorithm would next try panic mode. In the latter case, one of the repairs found to be viable would be applied.

The local recovery algorithms next to be considered were all derived from the local recovery algorithm for simple precedence parsers proposed by Graham and Rhodes [GR75]. Like Lévy, Graham and Rhodes use the terms *forward move* and *backward move* to name actions of their algorithm. However, the actions named by those terms are different for the two algorithms.

The Graham-Rhodes algorithm is divided into two phases: a *condensation phase* followed by a *correction phase*. Suppose an error has been detected. During the condensation phase, information is gathered from the context surrounding the error's detection point. The first step of the condensation phase is the *backward move*. So long as the top of the parse stack contains a handle, that handle is reduced. Because simple precedence grammars are uniquely invertible, the nonterminal symbol to which the handle is to be reduced is uniquely determined by the handle. Therefore, only one sequence of reductions will be possible. The backward move is followed by the *forward move*. The forward move parses the text following the error's detection point. The forward move continues until either a second error is detected, or the only possible parsing action is a reduction involving symbols to the left of the error's detection point.

The correction phase decides how to patch the error. The recoveries performed by the Graham-Rhodes algorithm consist of replacing a portion of the parse stack by the rhs of a rule. For each potential recovery, a check is made to ensure that parsing will be able to continue if that recovery is applied. The cost of each recovery found to be viable is then evaluated. Any recoveries whose costs exceed a predetermined threshold are rejected. If the costs of any recoveries fall below the threshold, the recovery whose cost is the lowest is applied.

The Graham-Rhodes algorithm cannot easily be adapted to work for LR parsers. For LR parsers, it is neither feasible nor necessary to do a backward move. Because LR grammars need not be uniquely invertible, there will usually be many different sequences of reductions that could be done for a given configuration of the parse stack. However, the purpose of a backward move is to gather information from the left context of an error's detection point. For an LR parser, much of that information is built into the parser's states. Therefore, doing a backward move would produce little information.

The forward move poses a more serious problem. LR parsers use an unbounded amount of left context information to help decide which parsing actions to perform. It is impossible to know what the left context of text following an error's detection point should be. However, because a forward move never does a reduction involving symbols to the left of an error's detection point, the left context of text following an error's detection point can be regarded as simply a state of the parsing automaton. Therefore, the number of contexts from which parsing may continue is bounded by the number of states of the parsing automaton.

The local recovery algorithm proposed by Mickunas and Modry [MM78] is a relatively straightforward adaptation of the Graham-Rhodes algorithm. Mickunas and Modry solve the forward move problem by performing multiple forward moves. Let a be the symbol immediately to the right of an error's detection point. The condensation phase of the Mickunas-Modry algorithm starts by identifying the set Ψ of states that permit shifts over a. For each state q in Ψ, the algorithm parses the text to the right of the detection point starting from state q. Each parse continues until either a second error is detected or the next parsing action is a reduction involving symbols to the left of the detection point. The sequence of states that is the result of a forward move is called a *recovery candidate*. Recovery candidates that are the results of parses that ended because the next action would be a reduction across the error's detection point are *correction candidates*. Recovery candidates that are the results of parses that ended because of later errors are *holding candidates*. Both the correction candidates and the holding candidates are sorted according to the number of tokens shifted before their trial parses halted.

The correction phase of the Mickunas-Modry algorithm is quite different from that of the Graham-Rhodes algorithm. The Mickunas-Modry algorithm attempts to find the recovery candidate that best matches the parse stack. It invokes a correction algorithm that tests various repairs for each correction candidate in order, starting with the candidate that shifted over the most tokens. The correction algorithm returns either a possible repair along with a cost for applying that repair, or an indication that no repair was found. If any suitable repairs were found, the lowest cost repair is applied, and normal parsing resumes. If none of the correction candidates yielded a repair, the error recovery algorithm is recursively reinvoked for each holding candidate in an attempt to repair the error that caused the forward move to terminate. The correction algorithm is then invoked for each holding candidate whose forward move was repaired. If any of the holding candidates can be repaired, the best repair is applied, and normal parsing resumes. If none of the recovery candidates leads to a repair, the recovery algorithm fails.

The correction algorithm used by Mickunas and Modry takes two arguments: a parse stack and a recovery candidate. The algorithm attempts to find the lowest cost repair that bridges the gap between its arguments. The only repairs considered are insertions and deletions of single tokens. The algorithm tries insertions first. If there are any tokens such that inserting one of them between the parse stack and the recovery candidate permits parsing to continue, the least costly of those insertions is returned. If no such insertion is found, attempts are made to repair the error further down in the parse stack. Let a be the accessing symbol of the state at the top of the parse stack. Let $\Psi = \{ q \mid g(q, a) = p \}$, where g is the parser's goto function, and p is the state at the start of the current recovery candidate. For each $q \in \Psi$, the correction algorithm is recursively reinvoked. The parse stack passed to the new invocation is the current parse stack minus its topmost element. The recovery candidate is the sequence of states formed by prepending q to the current recovery candidate. If any viable repairs are found, the lowest cost repair is returned as the result of the correction algorithm. Otherwise, deletions are attempted. Again, let a be the accessing symbol of the state at the top of the parse stack. If a is a terminal symbol, the stack is popped, and the correction algorithm is recursively reinvoked. If a is a nonterminal symbol, the parse stack is popped and then the string from which a was produced is reparsed ignoring the last token. The correction algorithm is then reinvoked. If deletion fails to produce a repair, the correction algorithm fails.

The fundamental problem of the Mickunas-Modry algorithm is its inefficiency. Performing multiple forward moves is a major source of inefficiency. Study of the Mickunas-Modry algorithm when applied to the Ripley-Druseikis test suite [RD78] revealed that few forward moves continued for more than a few symbols. The number of forward moves that were done, however, was surprisingly large. Consider the statement

$$quo := i \ over \ j;$$

The likely error is that the identifier *over* was mistakenly used in place of the keyword **div**. The parser detects the error between the identifiers *i* and *over*. The condensation phase carries out forward moves starting with the identifier *over* for every state of the parsing automaton that permits a shift over an identifier. There are 102 such states (out of a total of 394 states) in the LALR(1) parser for Pascal used in this study. Because the next symbol is another identifier (*j*), every forward move immediately fails. Thus, every forward move becomes a holding candidate. For each of those forward moves, the error recovery algorithm is recursively invoked. Hence, for each of the original forward moves, another 102 forward moves are commenced. Most of those 10404 secondary forward moves produce correction candidates. Thus, after a few thousand applications of the correction algorithm, a repair will be produced. To be fair, it it must be admitted that this is an extreme case. Nonetheless, even for more common examples, the number of forward moves considered can be very large.

Another solution to the forward move problem was proposed by Druseikis and Ripley [DR76] and independently by Pennello and DeRemer [PD78]. They create an extended parsing automaton that achieves the effects of multiple forward moves by performing a single forward move. The basis of their methods is that all of the forward parses can be carried out simultaneously provided they all perform the same reductions at the same points in the parse. The Druseikis-Ripley algorithm and the Pennello-DeRemer algorithm differ in matters of detail only; the formulation below is based on the Pennello-DeRemer algorithm.

Let M be an LR(k) parsing automaton. Let q_1 and q_2 be any two states of M, and let x be any lookahead string. Let $\alpha_1 = f(q_1, x)$ and let $\alpha_2 = f(q_2, x)$, where f is the action function of M. Then q_1 and q_2 are *α-equivalent* over x if and only if $\alpha_1 =$ **error**, or $\alpha_2 =$ **error**, or $\alpha_1 = \alpha_2$.

The Pennello-DeRemer algorithm uses a *forward move automaton* (FMA) to carry out its condensation phase. For each LR(k) parsing automaton M, there is a unique FMA. The FMA is another LR(k) parsing automaton. The state set of the FMA is the power set of Q, where Q is the state set of M. The set Q is the initial state of the FMA. The goto function δ of the FMA is derived from the goto function g of M. For each state s of the FMA and for each symbol a, $\delta(s, a) = \{ q \mid \textit{for some } p \in Q, g(p, a) = q\}$. Similarly, the action function ϕ of the FMA is derived from the action function f of M. A state s of the FMA is *consistent* over a lookahead string x if and only if for all p, $q \in s$, p and q are α-equivalent over x. For each state s of the FMA and each lookahead string x, $\phi(s, x) =$ **error** if s is not consistent over x. The error action does not signify that a error has been detected; it is, rather, a signal that the FMA should cease parsing. Suppose s is consistent over x. If $f(q, x) =$ **error** for all states $q \in s$, then $\phi(s, x) =$ **error**. Otherwise, there must be a single action α such that $f(q, x) = \alpha$ for some $q \in s$. In that case, $\phi(s, x) = \alpha$. Although it may at first appear that the FMA would be too large to be practical, it should be noted that most of the states of the FMA are inaccessible. Also, many of the states of the original parsing automaton can be shared by the FMA. Pennello and DeRemer report that the number of extra states needed for the FMA is about 20-50% of the number of states in the original parsing automaton.

The Pennello-DeRemer algorithm assumes that the FMA is precomputed. When an error is detected, the condensation phase of the Pennello-DeRemer error recovery algorithm uses the FMA to parse the text following the error's detection point. The parse continues until either the FMA detects an error or an attempt is made to reduce across the point in the input string at which parsing using the FMA commenced. The result of the parse is the sequence of terminal and nonterminal symbols produced by the shifts and reductions performed by the FMA. Given that the FMA is an LR(k) parsing automaton, the time required for the forward move should be roughly equivalent to the time required for normal parsing. The only extra operations that need to be done are the checks to see if a reduction crosses the error's detection point.

The condensation phase of the Pennello-DeRemer error recovery algorithm does not produce as much information as that of the Mickunas-Modry algorithm. The condensation phase of the Mickunas-Modry algorithm produces the set of sequences of states resulting from every parse that could possibly follow the detection point. The forward move performed by the FMA of the Pennello-DeRemer algorithm, on the other hand, produces only the sequence of terminal and nonterminal symbols that would be the common result of all parses starting from the detection point that do not lead to error configurations. As a result, the correction phase of the Pennello-DeRemer algorithm has less information available to it when selecting a repair. The correction phase, therefore, must test each possible repair it considers in ways that would be unnecessary for the Mickunas-Modry algorithm. Because each recovery candidate considered by the Mickunas-Modry algorithm is the result of parsing the string following the detection point starting from a given state, the correction phase need only check that a repair will cause the context to the left of the detection point to allow a transition into that state. The Pennello-DeRemer algorithm, on the other hand, must check that any possible repair will allow the parse to continue over the sequence of symbols produced by the FMA.

Before trying to repair any errors in a program, the entire program following the first error detected is parsed by the FMA. If the FMA halts before reaching the program's end, it begins parsing again starting from the symbol following the last symbol it was able to shift. The correction phase of the algorithm then tries to find corrections that will transform the sequence of strings produced by the FMA into a sentential form. The correction phase tries three types of repair: insertion, deletion, and replacement of single tokens. The repairs are first applied at the point at the detection point of the first error. A repair *succeeds* if it permits the parser to shift over a predefined number of symbols. If none of the repairs tried at the detection point succeeds, the algorithm tries applying the repairs to the symbols to the left of the detection point in right to left succession until either a repair is found or the algorithm reaches the leftmost end of the string. After repairing the first error, the algorithm then continues parsing until it either accepts the input, or detects a subsequent error. If another error is detected, the correction phase is again applied.. Note that there is no need to invoke the condensation phase again since the entire string has already been parsed by the FMA.

The most recent error recovery algorithm for LR parsers was first proposed and implemented by Feyock and Lazarus [FL76]. Graham, Haley, and Joy [GHJ79] independently developed a more refined and more practical version of the algorithm. Burke and Fisher [BF82] subsequently made further extensions to it. Although this algorithm was first reported by Feyock and Lazarus, it has, perhaps unfairly, come to be known as the Graham-Haley-Joy algorithm.

The Graham-Haley-Joy algorithm might best be described as a "shotgun" method. For each error detected, several recoveries are considered. The lowest cost recovery that allows parsing to continue is selected as the one to be applied. The algorithm possesses a fixed repertoire of repairs such as insertions, deletions, and replacements. Whenever an error is detected, the potential repairs are considered one at a time. Copies are made of the parse stack and a segment of the remaining input. Those copies are modified to reflect the repair being tested, and then they are parsed. The modified segment of the remaining input is called the *test string*, and the parse is called a *forward move*. (Note that "forward move" here means something different from either of the two previous uses of the same term.) The parse is halted when a new error is detected, the input string is accepted, or the parser shifts over the entire test string. Each repair is assigned a cost based on the nature of the repair. In addition, if the parse of the test string ends with an error, the cost of the repair is increased. If any suitable repairs are found for the given error, the lowest cost repair is applied and normal parsing resumes. Otherwise, the recovery algorithm resorts to panic mode.

A straightforward implementation of the Graham-Haley-Joy algorithm may do a great deal of redundant parsing. For example, consider the code fragment

```
i := i j;
if x < 0 then S;
```

The error here is the pair of adjacent identifiers on the first line. Many of the possible repairs will permit parsing to continue through the if-statement on the second line. Hence, unless the test strings are made unreasonably short, at least a portion of the if-statement will be parsed repeatedly as potential repairs are tested. However, there is exactly one state that contains a shift over the token **if** in the LALR(1) parser for Pascal used in this work. Therefore, for that parser, every parse of the if-statement will yield exactly the same result.

A way of implementing the Graham-Haley-Joy algorithm that avoids most redundant parsing has been developed as a part of this work. Let $a_1 \ldots a_m$ and $b_1 \ldots b_n$ be two strings such that $a_i \ldots a_m = b_j \ldots b_n$. Suppose that the state of the parser just before shifting over a_i is the same as its state just before shifting over b_j. Let k be the height of the parse stack just before the parser shifts over a_i. Then the actions of the parser after shifting over a_i will be identical to its actions after shifting over b_j until it performs a reduction that reduces the height of the parse stack (before pushing the new state onto the stack) to less than k. That last reduction is the *freeing reduction* of the initial shift. The net result of the parsing actions performed from the time the parser shifts over a given token until it performs the freeing reduction associated with that shift consists of popping some states off the parse stack, advancing over some input tokens, and then shifting over a nonterminal symbol. These facts can be exploited to avoid redundant parsing while testing potential repairs.

The new technique for implementing the Graham-Haley-Joy algorithm relies on information recorded during earlier forward moves to avoid redundant parsing during the current forward move. Whenever a forward move is about to shift over a token of its test string that is to the right of any changes made to the original string, a check is made to see if any previous forward moves shifted over that token starting from the same state. If so, a record will have been kept of the net result of the shift and all subsequent actions up to and including the associated freeing reduction (if one was performed). Therefore, that result can be implemented directly without having to reparse any portion of the input. Three forms of results may have been recorded, namely that the parser

1. shifts over the remaining characters of the test string,
2. detects an error k tokens later, or
3. pops k states off the parse stack, advances over n tokens of the test string, and shifts over the nonterminal symbol A.

In either of the first two cases, the forward move is terminated. In the last case, the stack and the remaining input are modified as indicated and then the forward move is resumed. Consider the previous example again. The first forward move that shifts over the token **if** will record that the net effect of the shift and the subsequent actions is to pop one symbol off the parse stack, advance the input 6 tokens, and shift over the nonterminal symbol "unlabeled statement." (Note that the recorded result depends on the particular parser being used; for other parsers, other results would be recorded.) All subsequent forward moves that would otherwise have to reparse the if-statement can now simply implement the recorded result (recall that there is only one state that contains a shift over the token **if**).

The information needed about the results of particular groups of parsing actions can be gathered without significantly slowing the parser. Whenever the parser shifts over a token in a test string that is to the right of all changes made to the string, an ordered triple is pushed onto an auxiliary stack. The form of the triple is (q, ℓ, h), where q is the state of the parser just before shifting, ℓ is the location of the token in the original input string, and h is the height of the parse stack after shifting. Whenever the parser performs a reduction that pops more than two states off the parse stack, the contents of the auxiliary stack are examined starting from the top of the stack. Let k be the height of the parse stack after the symbols have been removed, let n be the location of the current lookahead symbol in the original input string, and let A be the nonterminal symbol on the lhs of the rule used in the reduction. For each element (q, ℓ, h) of the auxiliary stack such that $h < k$, record that whenever the parser shifts over the symbol at location ℓ starting from the state q, the net result is to pop $k - h$ states off the parse stack, advance the parse $n - \ell$ tokens, and shift over A. Each such element is then popped off the auxiliary stack. Whenever the parser detects an error or shifts over the final token in the test string, the corresponding results are recorded for every element that remains on the auxiliary stack, and then the stack is emptied.

4.2 Applying Semantics to Repairs

The main reason for applying semantic data to error recovery is to produce better error diagnoses. Given a set of potential recoveries, those recoveries that lead to semantically correct programs should normally be preferred over those that do not. Every error repair algorithm must have a method for choosing among possible repairs. That selection mechanism is the obvious point at which to apply semantic data.

Lévy's algorithm uses two different mechanisms for choosing a repair. The forward move phase of the algorithm rejects all repairs that involve more than a predefined number of changes to the input. After the forward move phase, some other mechanism is needed to choose among the remaining repairs. There seems little point in applying semantics to the forward move phase. The cost of performing semantic analysis during the myriad possible forward moves is too great, and the benefits of doing so are too small. Any repairs that would be weeded out because of semantic information can also be rejected later. Semantic checking could be applied just after the forward move phase. If any of the repaired strings are found to be semantically correct, those repairs that do not produce semantically correct strings can be rejected. The syntactic costs of the repairs could then be used to select a repair from among the remaining possibilities. If none of the repaired strings are semantically correct, the choice of a repair could be based on a function of the semantic and syntactic costs of each potential repair.

Lévy's algorithm is thus easily extended to use semantic data. However, minimal-distance repair algorithms such as Lévy's are inherently slow. The order complexity of those algorithms is equivalent to the order complexity of general context-free parsing. In his paper, Lévy suggests some heuristic limits which can be imposed to reduce the order complexity of his algorithm. If those limits are applied, the order complexity becomes linear. Even with those restrictions, however, the algorithm is too slow to be considered practical.

The local recovery algorithms for LR parsers that were based on the Graham-Rhodes algorithm cannot easily be adapted to take advantage of semantic data. Semantic analysis cannot be done during the forward move because semantic actions that might critically affect the analysis may not have been executed. Furthermore, it cannot be done after the forward move, because the text following the detection point will already have been parsed. One way around this dilemma is to save copies of the tokens in the text parsed by the forward move. Semantics could then be applied to the selection of each repair. When selecting a repair, the text to the right of the detection point of that error could be reparsed and the semantic cost associated with the repair could be determined. After the repair is selected, the repaired text could be parsed so that semantic analysis could continue up to the next error.

The difficulty of adding semantics to the Graham-Haley-Joy algorithm depends on the way in which the forward move was implemented. If the forward move consists of parsing each test string independently, it is easy to extend the algorithm to make use of semantics. During each forward move, the semantic actions and checks associated with each reduction performed can be executed. The semantic cost thus determined can then be used to in computing the cost of the repair. This is the technique used in the Pascal auditor created as a part of this work.

An implementation of the forward move that avoids redundant parsing makes it harder to compute the cost of a repair. However, it may reduce the time spent computing those costs. If any redundant parsing is to be avoided, it is necessary to know which semantic actions can alter the contents of the symbol table or other global semantic entities. If the text parsed during a forward move executes any of those actions, the mechanism for avoiding redundant parsing should cease to be used for the rest of the forward move. The results of a parse should not be recorded if any of such action was performed during the parse. Whenever a result of a parse is recorded, the semantic cost associated with that result should also be recorded. Whenever a reduction to a nonterminal symbol is recorded, the semantic value computed for that symbol should also be recorded. Thus, in those contexts that are not semantically sensitive, both redundant parsing and redundant semantic analysis can be avoided.

The Graham-Haley-Joy algorithm is thus shown to be the most suitable basis for semantics-directed repairs. Lévy's algorithm and the Mickunas-Modry algorithm are too slow to serve as a basis for a practical error repair algorithm. The Pennello-DeRemer and Druseikis-Ripley algorithms are both fast enough to be used for syntactic error recovery, but are not easily modified to take advantage of semantic data. The Graham-Haley-Joy algorithm is both fast enough to be practical and can easily be modified to make use of semantic data.

4.3 Semantics for Semantics-directed Repairs

The analysis phase of a modern compiler consists mainly of performing a syntax-directed translation. The result of analyzing an input program is the same program expressed in another form. That form might be anything from a parse tree to absolute machine code. The transformation is effected by action routines associated with the syntactic constructs of the language. If the compiler's parser is produced from a grammar, each action routine is associated with a rule of the grammar. This section examines the characteristics the action routines must possess to support semantics-directed error recovery. The ways those routines are implemented is not discussed here; that topic is deferred until Chapter 5.

The action routines can affect error recovery in two ways. If an action routine finds a semantic error that may be the result of a syntactic error, it can signal that the syntactic repair algorithm should be invoked. It can also provide information to the error repair algorithm about the semantic costs of potential repairs.

Culik [Cul69] and Koster [Kos71] each proposed separating the checks for semantic errors from the rest of the action routines. Thus, each action routine is split into two parts: a semantic check and a semantic action. The semantic checks are predicates based on the attributes of the symbols named in the associated rules. During semantic analysis, the semantic check is evaluated before executing the corresponding semantic action. The check returns false if a semantic error is detected. As was noted in Chapter 3, Schmauch [Sch82] uses this scheme as a basis for semantics-directed repairs. Whenever a repair is evaluated, it is assigned a cost based on the number of tokens shifted before a semantic error is detected. Schmauch's algorithm makes no provision for invoking the syntactic error recovery algorithm in the event of a semantic error. The syntactic recovery algorithm could simply be invoked whenever a semantic error is detected. However, for many types of semantic errors there is no reason to suspect that the error is the result of a syntactic mistake. For example, it would be wasteful to invoke the syntactic repair algorithm whenever a semantic check detects an undeclared identifier, since there is little hope that such an error could be corrected by a syntactic repair.

The scheme used in this work for providing semantic data to the parser and the error recovery algorithm is based on an extended notion of semantic checks. Those extended semantic checks are called *guards*. Each semantic action can be preceded by a guard. A guard sets two global variables: a flag indicating whether the syntactic repair algorithm should be invoked, and a integer indicating the cost of performing the associated semantic action. The compiler writer decides which semantic errors will cause the flag to be set. Thus, errors such as undeclared identifiers need not cause the syntactic repair algorithm to be invoked. A guard must set the cost to zero if no semantic error is found; otherwise, it must be assigned a positive value.

The semantic actions must be able to cope with semantic errors. Error messages for semantic errors are issued by the semantic actions. A semantic action will be invoked even if the preceding guard signals that there is a semantic error. The syntactic repair algorithm executes the semantic actions while testing potential repairs (after setting a flag that blocks error reporting). If no syntactic repair that fixes the semantic error is found, the semantic action is invoked to issue an error message and generate an appropriate result.

It is sometimes possible to produce the correct semantic result in the presence of semantic errors. For example, the type of a relational expression is Boolean regardless of the types of its operands. Therefore, the semantic action for a relational expression can return a value indicating that the expression's type is Boolean even if the types of the expression's operands clash. This particular example has led to obviously better repairs in some instances. It is, of course, usually impossible to produce correct semantic results in the presence of semantic errors. In those situations, the semantic action should return a special error value as its result. The error value could be propagated by later rules to prevent detection of spurious errors.

The scheme outlined above was used in the Pascal auditor. That experience has shown that the scheme could be used as a basis for semantics-directed error recovery for a real programming language. However, it also exposed a basic flaw of the scheme, namely that the time required to analyze correct programs is significantly increased over the time required by conventional compilers.

One cause of that increase is that the guards and semantic actions must often check the same conditions. There is no communication between the guards and the semantic actions. The guards check for semantic errors and decide how they should be handled. However, it is the semantic actions that must produce the error messages. To know whether a message should be issued, the semantic action must usually perform the same checks as the guard. Programming languages with many data types and strong type checking require elaborate tests for semantic correctness. Examination of the run-time actions of the Pascal auditor has shown that those checks usually had to be performed twice. While the execution time of each semantic check is small, the time spent performing the redundant checks is a significant fraction of the total time required to analyze correct programs.

Another source of the implementation's inefficiency is that a single copy of the guards and actions is used by both the parser and the error recovery algorithm. Therefore, evaluating a guard or executing an action always involves a subroutine call. The Yacc parser generator [Joh78] is able to avoid this overhead by expanding the action routines inline within the parser itself. The overhead could have been avoided in the implementation by expanding the guards and actions inline within both the parser and the error recovery algorithm. The only objection to that expansion is the additional code space required. The total size of the guard and action routines is just under 11,000 bytes. Making duplicate copies of those routines would be undesirable for a small machine, but it would be acceptable for most paging machines (particularly since the copy created for the error recovery algorithm will not have to be loaded into memory unless an error is detected).

The semantic routines can be organized so that no redundant checks are performed for correct programs. In the previous scheme, the guards always produced both a flag value and a cost. The flag value serves only to trigger the repair algorithm; while the repair algorithm is executing, the flag value is not inspected. The cost, on the other hand, is never examined during normal parsing. Thus, the flag value and the cost are never needed at the same time. This observation suggests that it would be better to split the action routines into separate versions for normal parsing and for error recovery than to split them into semantic checks and semantic guards.

The new scheme requires three versions of each action routine. The first version is used during normal parsing, the second is used to test potential repairs, and the third is used to get semantic analysis back on track after recovering from an error. The versions of an action routine are similar except for the way in which they treat semantic errors. When the first version detects an error, it either sets the flag that signals that the syntactic repair algorithm should be invoked and returns, or it issues an error message and continues executing. When the second version detects an error, it assigns a cost to that error and continues; it never issues an error message. When the third version detects an error, it issues an error message and continues executing; it does not affect either the flag or the cost.

Since the new scheme requires different versions of each action routine for each function that those routines serve, each occurrence of an action routine should be expanded inline. The objections to inline expansion of the action routines under the previous scheme are exacerbated under the new scheme, since now three copies of each routine are needed. There are, however, reasons to believe that the space requirements will not increase as much as might at first be expected. Under the previous scheme, the codes for the guards were usually similar to the codes for the semantic actions but different enough that they could not be shared. Under the new scheme, the codes for the various versions of the action routines will usually be identical except for the portions for handling errors. Thus, there should be more chances for sharing code.

5
A Model of Compilation for Semantics-directed Error Recovery

Semantics-directed error recovery requires unusually tight linkage between parsing and semantic analysis. Many compilers defer semantic analysis until after parsing has been completed. Even one-pass compilers commonly delay semantic checking for each statement until after the entire statement has been parsed. Delaying semantic checking can result in inferior recoveries. Consider, for example, the statement

$$writeln(x.\ y)$$

where x and y are real variables. The apparent error is that a period has been used where a comma was intended. The semantic routines could detect this error before the parser shifts over the period. If that happens, a semantics-directed error recovery algorithm would most likely replace the period with a comma. However, the statement is syntactically correct since $x.\ y$ is a well-formed record selection. Therefore, if semantic checking is deferred, the error will not be discovered until after the statement has been parsed. By that time, only a backtracking error recovery algorithm could find the best recovery. The backtracking error recovery algorithms that have been proposed thus far are too slow to be practical. Therefore, it must be assumed that deferring semantic checking would lead to inferior recoveries.

Semantics-directed error recovery also requires the ability to undo the effects of semantic operations. When testing a potential recovery, the error recovery algorithm must evaluate the semantics associated with that recovery to determine its semantic cost. If the recovery is rejected, the semantic operations done while testing it must not affect later stages of compilation. Semantic operations performed by conventional compilers are not easily reversed. The effects of a executing a semantic action can include altering global variables, updating attributes in the semantic stack, and inserting and deleting symbol table entries. Since semantic operations can cause so many kinds of changes, a general history mechanism would be needed to make it possible to reverse their effects. While such a mechanism could be implemented, the associated time and space overheads are daunting. Therefore, a more restricted compiler organization is needed.

This chapter explores some ways of organizing a compiler to support semantics-directed error recovery. First, a paradigm of reversible semantics based on attribute grammars is given. Restricted forms of attribute grammars that could support semantics-directed error recovery are considered. The chapter concludes with the presentation of a model of compilation that can support semantics-directed error recovery and yet is efficient enough to be used as a basis for practical compilers.

5.1 Attribute Grammars

Attribute grammars are currently the most popular model of the analysis phase of compilation. Analyzing a program according to an attribute grammar consists of generating an evaluation for it. Producing an evaluation involves two distinct steps. First, the program's parse tree must be constructed. Then, the attributes of the parse tree's nodes are evaluated. Every attribute of every node in the parse tree is initially considered to be unknown. For each attribute of a node, the associated dependency vector determines which other attributes must be assigned values before it can be evaluated. So long as any attributes have not been assigned a value, an attribute that can be evaluated is found, the associated semantic function is evaluated, and the resulting value is assigned to the attribute. This process continues until either every attribute has been assigned a value, or no further attributes can be evaluated because of circular dependencies.

The attribute grammars used by compiler generation systems usually permit terminal symbols to possess "inherent attributes." The values of the inherent attributes are set by the lexical analyzer. For example, the inherent attribute of an identifier might be the string representing that identifier, while the inherent attribute of an integer constant might be its value. Inherent attributes are theoretically unnecessary. The inherent attributes of a symbol could be replaced by synthesized attributes if the underlying grammar for the language were extended down to the character level. The additional time required to parse according to such a grammar renders that possibility impractical.

Attribute grammars represent a more restrictive model of compilation than is embodied by most compilers. The only semantic operation is to evaluate attributes. Further, once an attribute has been assigned a value, its value cannot be changed. Finally, there are no global data structures. Those restrictions make it easy to reverse the effects of semantic operations.

One concrete implementation of reversible semantic analysis for attribute grammars involves pairing each attribute with a pointer variable. Attribute evaluation requires a means of indicating which attributes have already been assigned values. In this implementation, the pointer variables indicate whether the associated attributes have been evaluated. Initially, the pointer variables are all set to null. Any attribute whose associated pointer variable is null is considered to be unevaluated. When an attribute is assigned a value, its pointer variable is set to the address of the pointer variable paired with the last attribute that was previously assigned a value. A special flag value must be assigned to the pointer variable associated with the first attribute to be evaluated. Thus, the pointers paired with evaluated attributes form a singly linked list ordered in the reverse of the order of evaluation. The effects of all semantic operations done after a given time can be undone simply by following the chain of pointer variables back to last one that had been set at that time, resetting each pointer variable encountered along the way to null.

5.2 LL- and LR-attributed Grammars

Although general attribute grammars allow the effects of semantic operations to be reversed, they do not provide a suitable basis for semantics-directed error recovery. For general attribute grammars, the entire parse tree of a program must be constructed before any of that tree's attributes can be evaluated. Therefore, no semantic checking can be done before parsing is completed. Thus, it is impossible to use semantic information to recover from syntax errors. Some forms of attribute grammars that do permit semantic analysis to be linked with parsing have been proposed. Two of the most powerful forms, the LL-attributed grammars [LRS74] and the LR-attributed grammars [Wat77], are considered in this section.

For attribute grammars, performing semantic analysis while parsing means that whenever the parser advances over an input token, all of that token's attributes must be evaluated. Also, whenever the parser reaches the end of the rhs of a rule in the derivation produced by the parser, the attributes of the symbol on the lhs of that rule must be evaluated. Evaluating synthesized attributes while parsing poses no problem. By the time the parser reaches the end of the rhs of a rule $X_0 \rightarrow X_1 ... X_n$ that is in the derivation produced by the parser, the attributes of the symbols on the rhs of the rule must all have been evaluated. Therefore, any synthesized attributes of the symbol on the lhs of the rule can be evaluated. Thus, only inherited attributes require special handling.

Lewis, Rosenkrantz, and Stearns [LRS74] show that for an L-attributed grammar the attributes of a parse tree can be evaluated in a single left-to-right top-down traversal of the tree. Since such a traversal corresponds to the order in which the nodes are encountered during a left-to-right top-down parse, an L-attributed grammar permits semantic analysis while parsing if the underlying grammar is LL(k). An L-attributed grammar whose underlying context-free grammar is LL(k) for some k is an *LL-attributed grammar*. An automaton for evaluating LL-attributed grammars while parsing is described in [LRS74].

Methods for implementing semantic evaluation while parsing for use with bottom-up parsers have also been proposed [LRS74, Wat77, Poh83]. The method described here is based on the method proposed by Watt [Wat77]. Watt's method is defined, not for attribute grammars, but for a related class of grammars called ***affix grammars*** [Kos71]. The variant described below is an adaptation of Watt's method for L-attributed grammars. Given an L-attributed grammar AG, a total order is defined over the inherited attributes of each symbol. The values of the currently relevant inherited attributes are maintained in a global stack called the ***inherited attribute stack***. Whenever a rule $X_0 \rightarrow X_1 ... X_n$ is used in a reduction, the values of the inherited attributes of X_0 will appear in order at the top of the inherited attribute stack. Let G be the underlying context-free grammar of AG. A new grammar, called the ***head grammar***, is created from G. The head grammar contains new symbols and rules that are used to manipulate the inherited attribute stack. A ***control rule*** is a λ-rule whose associated action routine modifies the inherited attribute stack. The nonterminal symbol on the lhs of a control rule is a ***control symbol***. The control symbols must be symbols that do not appear in the vocabulary of G. Control symbols are used to evaluate inherited attributes and to maintain those values in the proper order on the inherited attribute stack. Let $X_0 \rightarrow X_1 ... X_n$ be a rule of G. The corresponding rule of the head grammar is created by adding control symbols to the immediate left and right of those symbols that require modification of the inherited attribute stack. For each X_k, $1 \leq k \leq n$, the semantic functions and dependency vectors used to compute the values of the inherited attributes of X_k are examined. If the compiler generator can determine that the values of the inherited attributes of X_k must equal the values already at the top of the inherited attribute stack, no control productions are added around X_k. Otherwise, X_k is surrounded by a pair of control symbols. The action routine associated with the control symbol to the left of X_k pushes the values of those inherited attributes onto the inherited attribute stack. The action routine associated with the control symbol to the right of the symbol pops those values off the stack. Note that different orderings of the inherited attributes may result in different head grammars.

An L-attributed grammar whose head grammar is LR(k) for some ordering of the inherited attributes is an *LR(k)-attributed grammar*. An L-attributed grammar that is an LR(k)-attributed grammar for some k is an *LR-attributed grammar*. It is difficult characterize the LR(k)-attributed grammars. If the underlying context-free grammar G of an attribute grammar is not LR(k), then its head grammar will not be LR(k). However, the control symbols and control productions that are added to the head grammar may cause it to fail to be LR(k) even when G is LR(k). Also, the quality of the algorithm used to produce the head grammar affects which L-attributed grammars are LR(k)-attributed grammars. For example, a variety of tests could be used to determine whether the inherited attribute stack must be modified. A compiler generator that uses a powerful test will produce LR(k) head grammars more often than one that uses a weaker test.

Attribute grammars for programming languages normally use inherited attributes to move information about the environment in which symbols occur down to the instances of those symbols in the parse tree. Thus, for most parts of the context-free grammar of a programming language, the head grammar will be the same as the original grammar. Consider, for example, the piece of an attribute grammar for expressions shown in Figure 5.1. The inherited attribute ENV is an environment, the synthesized attribute DEF is a definition, and the inherent attribute TAG is a string. An environment is assumed to be a list of definitions for symbols. The function *lookup* takes a string and an environment as its arguments. If the string is associated with a definition, that definition is returned; otherwise, an error value is returned. Whenever a reduction is performed according to one of the rules given above, the topmost value on the inherited attribute stack will be the current environment value. A compiler generator should be able to determine that the stack need not be modified for any of the rules listed, since the semantic functions assigning values to the ENV attribute are all copy operations. Therefore, those rules should be copied into the head grammar without change.

Both LL- and LR-attributed grammars are well suited for semantics-directed error recovery. Because the attributes are evaluated in a single pass, there is no need to store the parse tree. For LL-attributed grammars, the attribute values can be maintained in a separate semantic stack. Whenever the error recovery routine is entered, a copy can be made of the contents of the semantic stack. The effects of semantic operations done while testing a potential recovery can be undone by copying back the original contents of the semantic stack. For LR-attributed grammars, the values of the inherited attributes of symbols that have not yet been shifted are stored in the inherited attribute stack. The synthesized and inherited attributes of the symbols that have been shifted can be stored in a separate semantic stack. The semantic stack for LR-attributed grammars would be maintained in parallel with the parse stack. As in the case of LL-attributed grammars, the values of the inherited attribute stack and the semantic stack could be copied and then restored as needed.

Unlike general attribute grammars, LL- and LR-attributed grammars make semantic information available when needed to evaluate the semantic costs of potential recoveries *provided* that the language being analyzed requires symbols to be declared before they are used. Languages with that property are called *one-pass languages*. Few languages are strictly one-pass languages. However, many languages, including Fortran [ANS78], Cobol [ANS74], Pascal [ANS83], and C [KR78], are nearly one-pass languages, *i.e.*, they are one-pass except with respect to a few constructs such as labels. Semantics-directed error recovery for those semantic features of a nearly one-pass language that are truly one-pass can be supported by LL- and LR-attributed grammars.

```
$expr_0 \rightarrow expr_1 + term_1$
        {
            $expr_1.ENV = expr_0.ENV;$
            $term_1.ENV = expr_0.ENV;$
            . . .
        }

$expr_0 \rightarrow term_1$
        {
            $term_1.ENV = expr_0.ENV;$
            . . .
        }

$term_0 \rightarrow term_1 * factor_1$
        {
            $term_1.ENV = term_0.ENV;$
            $factor_1.ENV = term_0.ENV;$
            . . .
        }

$term_0 \rightarrow factor_1$
        {
            $factor_1.ENV = term_0.ENV;$
            . . .
        }

$factor_0 \rightarrow identifier_1$
        {
            $factor_0.DEF = \mathit{lookup}(identifier_1.TAG, factor_0.ENV);$
            . . .
        }

$factor_0 \rightarrow ( expr_1 )$
        {
            $expr_1.ENV = factor_0.ENV;$
            . . .
        }
```

Figure 5.1 A sample LR-attributed grammar

5.3 A Practical Organization that Supports Semantics-directed Error Recovery

Despite the fact that LL- and LR-attributed grammars provide workable models of compilation for semantics-directed error recovery, the recovery techniques implemented as part of this work are based on an alternative model. The main reason for rejecting LL- and LR-attributed grammars was efficiency. Although much effort has been devoted to developing compiler generation systems based on attribute grammars, none of those systems produce implementations that are acceptably fast and only the most restrictive systems are acceptable in terms of storage requirements.

Another reason for rejecting LL- and LR-attributed grammars is that they are hard to use. It is pointed out in [LRS74] that it is difficult to write LL(k) grammars for real programming languages. While it is relatively easy to write LR(k) grammars for programming languages, the restrictions on semantic operations make it difficult to write LR-attributed grammars. The difficulty of writing attribute grammars for programming languages is possibly the main reason attribute grammar based compiler generators are not widely used. Table-driven parsing techniques gained acceptance before they began to match the efficiency of hand-coded parsers because it is easier write a grammar for a language than it is to hand-code a parser. Attribute grammars, on the other hand, are often more difficult to write than equivalent hand-coded semantic routines.

A major cause of the inefficiency of compilers based on LL- and LR-attributed grammars is that current implementations of attribute grammars do not model the functions of symbol tables efficiently. Symbol tables are normally designed to allow rapid insertion, deletion, and look up of symbols. Attribute grammars, however, do not permit the values of attributes to change once they have been set. Therefore, it is difficult to emulate a dynamic structure such as a symbol table without incurring large space or time overheads. The data structure most often used to implement symbol tables for attribute grammars is the inverted tree. An inverted tree is a directed tree in which the edges go from the leaves to the root. Inverted trees support rapid insertion and deletion of symbols, and require no more space than normal symbol tables. However, the average time required to look up a symbol in an inverted tree is proportional to the number of currently visible table entries. Therefore, compilers that use inverted trees to represent symbol tables are too slow for production use.

An obvious solution to the problem of efficiently modeling the functions of a symbol table is to add a conventional symbol table to an attribute grammar. That is the approach taken in this work. The semantic functions of the attribute grammar are extended to include semantic actions that use and modify the symbol table. The symbol table must be implemented in a way that permits symbol table operations to be undone. Two symbol table organizations that support reversible semantics are discussed in Section 5.4.

Once the decision was made to use a symbol table, it appeared that there would be no need for inherited attributes. The implementation of the Pascal auditor uses only synthesized attributes. Hence, it was possible to use an LALR(1) parser generator to produce the parser and semantic analyzer for the Pascal auditor. Since LALR(1) parser

```
fncall    →  fnpart ')'

fnpart    →  fnhead  expr

fnhead    →  fnname '('  |  fnpart ','

fnname    →  name

name      →  IDENTIFIER
```

Figure 5.2 Grammar for function calls without using inherited attributes

```
fncall      →  name '(' expr-list ')'

expr-list   →  expr-list ',' expr  |  expr

name        →  IDENTIFIER
```

Figure 5.3 Grammar for function calls using inherited attributes

generators are more common than attribute grammar based compiler generators, the techniques used in the Pascal auditor should be móre broadly applicable than techniques based on LR-attributed grammars. However, avoiding inherited attributes has its drawbacks. Some of the rules of the grammar had to be factored in unusual ways to permit semantic checking to be done while parsing. If the implementation had incorporated both a symbol table and inherited attributes, a simpler grammar could have been used. For example, the grammar for function calls used in the Pascal auditor is shown in Figure 5.2. The rules are factored so that type information obtained from the symbol table entry for the function name is available to the semantic routines for parameters. If the Pascal auditor had been implemented using an LR-attributed grammar, the more natural grammar for function calls shown in Figure 5.3 could have been used instead. Type information obtained from the function name could be passed to the semantic routines for parameters through inherited attributes.

The model organization of a compiler used in this work consists of an LALR(1) parser, a symbol table, and synthesized attributes. The values of the synthesized attributes are stored in a semantic stack which is associated with the parse stack. The ability to reverse the effects of semantic operations on the parse stack is implemented by copying and restoring the semantic stack as necessary. This model is not far different from the organization of a conventional one-pass bottom-up compiler. The major differences are that semantic actions are not allowed to reference or modify global variables except through symbol table operations, and that attributes cannot be modified once set.

The expressive power of this model of compilation is less than that of the LR-attributed grammars. Therefore, the set of languages for which semantic analysis can be linked with parsing under this model of compilation is a subset of the languages for which semantic analysis can be linked with parsing using LR-attributed grammars. Hence, this model of compilation is satisfactory only for languages that are nearly one-pass languages.

5.4 Symbol Tables

Special symbol table capabilities are needed to support semantics-directed error recovery. If any changes are made to the symbol table while testing a potential recovery, it must be possible to undo those changes. The ability to undo symbol table operations that were done during normal compilation can also prove useful. This section describes two symbol table organizations that allow symbol table operations to be undone. The first organization permits only those operations done while testing recoveries to be undone. The second allows the symbol table to be backed up to any previous state.

Most modern programming languages are block-structured. In a block-structured language, scopes of declarations are tied to syntactic constructs called *blocks*. The blocks are strictly nested. Four symbol table operations are needed to be able to compile a strictly block-structured language, namely

1. look up the current definition of a symbol,
2. insert a new definition for a symbol at the current nesting level,
3. increase the current nesting level, and
4. delete all definitions at the current nesting level and then reduce the nesting level by one.

Almost all programming languages include a few features that violate the rules of block-structuring. Additional operations may be needed to implement those features.

One way of implementing symbol tables so that the effects of symbol table operations can be undone involves using two tables. The first symbol table is used during normal compilation. No semantic operations done while testing a recovery are allowed to affect the first table. During a test, the second table is used as a filter through which the effects of symbol table operations are implemented. Each entry in the second table contains a mark bit that indicates if the entry has been deleted. It is assumed that a special value called *undefined* is returned when an attempt is made to look up an undefined symbol. A global flag indicates when the compiler is testing a possible recovery. During normal compilation, the first table is used for all operations. Before testing a recovery, the global flag is turned on and the second table is cleared. Any auxiliary variables used by the symbol table, such as the variable that records the current nesting level, should be copied so that they can be restored at the completion of the test.

The routine to look up definitions of symbols uses the first table during normal parsing. While testing a recovery, the routine first looks for a symbol in the second table. If the second table does not yet contain a copy of the symbol, the definition is obtained from the first table. The algorithm shown in Figure 5.4 implements the look up routine. *Lookup* places copies of definitions obtained from the first table into the second table instead of simply setting a pointer to the original definition because the semantic routines may modify the definition. If the compiler writer knows that a definition cannot be changed by later semantic actions, the second table can share the definition with the first table.

Inserting definitions and increasing the nesting level are both easily handled. If a recovery is being tested, the definition is inserted into the second table; otherwise it is inserted into the first. Increasing the nesting level does not require special handling; any semantic action can increase the nesting level.

The operation consisting of deleting all symbols at the current level and reducing the nesting level is called *popping the scope*. Popping the scope while testing a recovery poses some hard problems. Entries in the first table cannot simply be deleted since they will have to be restored at the end of the test. They could be unlinked from the table and yet saved for later restoration. However, that involves complicated pointer manipulations. One solution is to use the second table to mark those symbols that are no longer defined after popping the scope. Let *entry* be a function that takes an integer ℓ and a symbol s as its arguments. *Entry* returns the highest level entry among the set of entries for s that were entered at a nesting level less than or equal to ℓ. If the set of

```
function Lookup(s: symbol): definition;
begin
    if testing a recovery then
    begin
        if the second table contains an entry for s then
            if that entry is marked as deleted then
                return undefined;
            else
                return the definition of s from the second table;
        else
        begin
            if the first table contains an entry for s then
            begin
                make a copy of the definition of s in the first table;
                enter that definition into the second table;
                return the copied definition
            end
            else
                return undefined;
        end
    end
    else if the first table contains an entry for s then
        return the definition of s from the first table
    else
        return undefined
end
```

Figure 5.4 The look up algorithm

entries is empty, *entry* returns *undefined.* Then popping the scope can be implemented by the procedure shown in Figure 5.5.

The scheme described above makes restoring the symbol table after testing a recovery trivial. No symbol table operations performed during a test make any changes to the first table. After testing a recovery, the global flag signaling that a test is being performed is turned off, all storage allocated to the entries in the second table is freed, and the global variables whose values were saved at the start of the test are reset to their previous values.

This symbol table organization requires some additional restrictions on semantic operations. Care must be taken not to rely on the addresses of definitions. If a semantic routine depends on the locations of definitions returned by the look up routine, there is a chance the routine will not work correctly while testing a recovery. If a definition must be copied by the look up routine, its address during normal compilation will not be the same as its address during testing. Also, there may be problems with shared data structures. The look up routine may make separate copies of a structure shared by two or more definitions. Therefore, if the semantic routines rely on changes to the structure affecting every definition that accesses it, they may fail while testing a potential recovery.

The efficiency of this table organization depends on the data structures used to represent the tables. The routine for popping the scope is particularly sensitive. If the tables are implemented using hashing with chaining where the chains can hold entries for

```
procedure PopScope;
var ℓ, n: integer;
begin
    if testing a repair then
    begin
        ℓ ← the current nesting level;
        n ← the nesting level when the test began;
        delete all entries in the second table
        whose nesting level is ℓ;
        if ℓ ≤ n then
            for each entry e in the first table
                whose nesting level is ℓ do
            begin
                s ← the symbol defined by e;
                if there is no entry for s in the second table then
                begin
                    e1 ← entry(ℓ, s);
                    if e1 = undefined then
                    begin
                        make an entry for s in the second
                        table at the lowest nesting level;
                        mark that entry as having been deleted
                    end
                    else
                        insert a copy of e1 in the second
                        table at the same nesting level as e1
                end
            end
    end
    else
        delete all entries in the first table
        whose nesting level is ℓ;

    the current nesting level ← ℓ - 1
end
```

Figure 5.5 The algorithm for popping the scope

different identifiers, then when a symbol is deleted, it will sometimes be necessary to scan an entire chain to discover that there are no lower level definitions of that symbol. A better representation is to use a name table with separate definition chains for each name. That representation makes it possible to determine if there are any lower level definitions of a given symbol in time proportional to the number of definitions. If the routines that copy entries from the first table to the second also create pointers back to the original entry, that condition could be checked in constant time.

The symbol table organization described above permits the effects of symbol table operations done during a test of a potential recovery to be undone. However, it does not permit operations done during normal compilation to be undone. An alternative organization that provides that capability has been developed. The alternative organization was intended for use with a backtracking error recovery algorithm. That algorithm was abandoned because it was too slow. However, the symbol table organization proved attractive in its own right and was used in the implementation of the Pascal auditor. Other reasons for wanting to be able to reverse the effects of symbol table operations done during normal compilation are discussed in Sections 6.5 and 7.5.

The alternative organization uses a specialized history mechanism. A counter called the *timeclock* is used to keep track of the sequence of symbol table operations. The timeclock is incremented whenever the symbol table's contents are altered. It is also incremented whenever the nesting level is increased. Each symbol table entry contains two special fields: the *entered* field and the *deleted* field. The two fields provide the information needed to back up the symbol table. The entered field is assigned the value of the timeclock at the time the entry is inserted. The deleted field is initially set to zero. Whenever an entry is deleted, the routine that performs the deletion checks if its deleted field's value is still zero. If so, the deleted field is assigned the current value of the timeclock. The reason the deleted field might not be zero is that the entry may have been deleted and then replaced. Deleted entries are not destroyed; they are placed on a list of deleted entries.

This symbol table organization requires little that is unusual from the routines implementing symbol table operations. The look up routine does nothing beyond its normal function. The timeclock must be incremented whenever the nesting level is increased. Whenever an entry is inserted, the timeclock must be increased and its value must be recorded in the entry's entered field. Popping the stack is a bit more involved. First, the timeclock is incremented. Then each entry at the current nesting level is removed from the symbol table and placed on the deleted entries list. If an entry's deleted field is zero, that field is assigned the current value of the timeclock. The entries on the deleted entries list are not copies; they are the original entries in their original locations.

The space requirements of this organization are greater than normal. The extra fields needed for each table entry will add a significant fraction to the size of the symbol table. The fact that symbol table entries are not deallocated when they are deleted also adds to its space requirements. However, there are many production compilers that also never delete symbol table entries. Multi-pass compilers seldom delete any symbol table entries until after the analysis phase of compilation is completed. Also, some compilers do not delete symbol table entries until after completing code generation so that they can produce memory maps for all named objects.

If the error recovery algorithm is to make use of the ability to undo symbol table operations done during normal compilation, it must have a way of matching configurations of the parser with configurations of the symbol table. Assume that the error recovery algorithm does backtracking. If the parse is backed up to an earlier configuration, the symbol table should be backed up to the configuration it had at the time the parser first entered that configuration. If the symbol table is not backed up to the proper configuration, further semantic analysis could result in spurious errors being indicated. One way of linking configurations of the symbol table and configurations of the parse stack is to record the timeclock and the nesting level each time the parser shifts over a symbol. The recorded values of the the timeclock are called the *timestamps* of the configurations. Given a timestamp and a nesting level, the symbol table can be backed up to the corresponding configuration.

The history mechanism makes it possible to back up the configuration of the symbol table either temporarily or permanently. A semantics-directed error recovery algorithm that performs backtracking needs the ability to temporarily back up the table to be able to test possible recoveries. Once a recovery is selected, it permanently backs up the symbol table and then implements the recovery.

Temporarily backing up the symbol table involves two routines: one to back it up, and one to restore it. The routine to back up the symbol table takes two arguments, the timestamp t and the nesting level ℓ for the configuration to be restored. The text of the back up routine is shown in Figure 5.6.

```
procedure Backup(t, ℓ: integer);
begin
    SavedTimeClock ← TimeClock;
    SavedNestingLevel ← NestingLevel;
    NestingLevel ← ℓ;

    Inactive ← nil;
    for each entry e in the symbol table such that e.entered > t do
    begin
        remove e from the symbol table;
        append e to Inactive
    end;

    for each entry e in Deleted such that e.entered ≤ t and e.deleted > t do
    begin
        remove e from Deleted;
        place e back in the symbol table
    end
end
```

Figure 5.6 The back up routine

The routine references some global variables. *TimeClock* is the name of the timeclock. *NestingLevel* is the variable containing the nesting level. *SavedTimeClock* and *SavedNestingLevel* are used to save the values of the timeclock and nesting level so that they can be restored later. *Inactive* is a global list of entries. It holds all entries dumped from the symbol table. *Deleted* is the deleted entries list. Note that the timeclock is not set to t.

The routine shown in Figure 5.7 restores the symbol table. The reason *Backup* did not reset the timeclock can now be explained. The symbol table entries added after backing up the table can be identified because their entered fields are greater than the old timeclock. Similarly, the entries that were deleted after backing up the table can be identified because their deleted fields are greater than the old timeclock.

Permanently backing up the symbol table is a comparatively simple operation. The algorithm for permanently backing up the symbol table is shown in Figure 5.8.

This symbol table organization requires that semantic operations be prohibited from modifying definitions obtained from the symbol table. Since copies of definitions are not made while testing recoveries, as was the case with the preceding symbol table organization, changes to definitions made while testing recoveries would not be undone after the test is completed. One benefit of not making copies is that the semantic

```
procedure Restore;
begin
    for each entry e in the symbol table such that
        e.entered > SavedTimeClock do
    begin
        remove e from the symbol table;
        free the storage allocated to e
    end;

    for each entry e in Deleted such that e.entered > SavedTimeClock do
    begin
        remove e from Deleted;
        free the storage allocated to e
    end;

    for each entry e in the symbol table such that e.deleted ≠ 0 do
    begin
        remove e from the symbol table;
        append e to Deleted
    end;

    for each entry e in Deleted such that e.deleted > SavedTimeClock do
    begin
        e.deleted ← 0;
        remove e from Deleted;
        place e back in the symbol table
    end;

    for each entry e in Inactive do
    begin
        remove e from Inactive;
        place e back in the symbol table
    end;

    TimeClock ← SavedTimeClock;
    NestingLevel ← SavedNestingLevel;
end
```

Figure 5.7 The restore routine

routines can rely on the definitions always being at the same addresses. That consistency is exploited in the type equivalence routines of the Pascal auditor.

The two symbol table organizations described above exhibit very different characteristics. The first organization adds little to the cost of symbol table operations performed during normal compilation. The only added cost is the time to test the global flag indicating that the compiler is not testing a recovery. The costs of performing symbol table operations while testing a recovery, however, are high. Further, the only space overhead of the scheme during normal compilation is the space required for the flag bit. While testing a recovery, extra space will be needed for the copies of the entries.

```
procedure Reset(t, ℓ);
begin
     TimeClock ← t;
     NestingLevel ← ℓ;

     for each entry e in the symbol table such that e.entered > t do
     begin
          remove e from the symbol table;
          free the storage assigned to e
     end;

     for each entry e in Deleted such that e.entered > t do
     begin
          remove e from Deleted;
          free the storage assigned to e
     end;

     for each entry e in Deleted such that e.deleted < t do
     begin
          e.deleted ← 0;
          remove e from Deleted;
          place e back in the symbol tab
     end
end
```

Figure 5.8 The reset routine

The history-based organization suffers the same time and space overheads during normal compilation as it does while testing a recovery. The overheads consist of maintaining the entered and deleted fields of the entries and maintaining the deleted entry list. Experience with the Pascal auditor has shown that the time spent backing up and restoring the symbol table before and after each test is small. In practice, testing a recovery usually does not involve making any changes to the symbol table, and so there is no need to restore the table.

The choice between the two organizations is simple so long as the ability to back up the symbol table is used solely to implement semantics-directed recovery. If the recovery algorithm uses backtracking, the history-based organization must be used. If it does not backtrack, the costs of using the history-based organization during normal compilation give the edge to the simpler organization. However, there can be other reasons for using the history-based organization. Section 6.5 shows how that organization could be used to implement a limited backtracking capability. Section 7.5 shows its advantages in implementing panic mode recoveries. The compiler writer must decide if those benefits outweigh the costs.

The two symbol table organizations described above are representative of many possible ways of implementing a symbol table suited for semantics-directed error recovery. Symbol table organizations that permit undoing symbol table operations done during normal compilation appear to require maintaining some data that is not needed for normal compilation. The time spent collecting that data will add to the compilation time of programs that are free of errors. Organizations that permit only operations done while testing potential recoveries to be undone are less flexible, but can be implemented with little or no impact on the time required to compile error free programs.

6
Erroneous Reductions

Errors often are not detected until after the parser has done some reductions based on the erroneous input. Consider the erroneous statement

$$x = 0.0$$

The apparent error is that the symbol '=' appears in place of the symbol ':='. Some parsers will not detect this error until after reducing x to a statement since, in Pascal, an identifier by itself is a syntactically correct procedure statement. Unless the error recovery algorithm is able to undo the effects of the erroneous reductions, it will be unable to find a good recovery. At best, it might report that a malformed statement has been detected.

The Pascal if-statement illustrates a harder problem. Consider the statement

$$\textbf{if } x < y \textbf{ then } min := x;\ \textbf{else } min := y$$

Pascal does not permit a semicolon to precede the keyword **else**. However, a natural grammar for if-statements would have the parser reduce the text to the left of the semicolon to a statement before shifting the semicolon. If that happens and there is no means of undoing the erroneous reductions, the likely response to this error would be to delete the keyword **else**. To avoid the erroneous reductions in this case, the parser would have to check two symbols of lookahead before doing a reduction.

There are two general methods for avoiding the harmful effects of erroneous reductions. Parsers can be constructed so that they are less likely to perform an erroneous reduction. Alternatively, it is possible to provide the ability to undo erroneous reductions. This chapter explores implementations of both techniques.

6.1 General Backtracking

Some authors [Lév75, FL76, MM78] have advocated using a general backtracking facility to solve the problem of erroneous reductions. Two methods of implementing such a facility have been suggested. One method involves building the derivation tree for each nonterminal symbol as it is recognized. When a error is detected, the trees provide the information needed to undo any reductions.

The other method for implementing general backtracking relies on reparsing portions of the input text. An index into the input text is maintained for each entry in the parse stack. The index for an entry references the first token in the text from which the entry was derived. Suppose that the configuration of the parser is to be backed up to the configuration it had before shifting some token t. First, the rightmost entry e in the parse stack whose index references either t or a token to the left of t is found. The index into the input text for e is saved, and then e and all entries to the right of it are popped off the parse stack. Lastly, the portion of the input text starting from the token referenced by the index that was saved and ending with the token immediately to the left of t is reparsed.

Backtracking and semantics-directed error recovery are compatible. However, if backtracking is used, the state of the parse and the state of semantic analysis must be kept consistent. Thus, if some reductions are undone, the effects of the semantic actions associated with them must also be undone. A compiler based on the model of compilation described in Chapter 5 that uses a history-based symbol table organization can efficiently reverse the effects of semantic actions.

The techniques for implementing general backtracking that have been proposed so far are too slow to be practical. The parse tree for a program typically occupies from ten to one hundred times as much space as the original input text. The time and space needed to construct derivation trees render the derivation tree based method impractical. The method based on reparsing does not suffer from serious space overheads. If there is not enough space to store the input tokens in main memory, they can be reread from the source file. The trouble with reparsing is the time involved. Suppose that an error is detected just after a large procedure has been recognized. Backing up the parse by one token would require reparsing almost the entire procedure.

Error recovery algorithms that perform general backtracking can have trouble with multiple errors. Suppose the recovery algorithm backs up over text created by an earlier recovery. If the algorithm makes a change to the input text to the left of the patched text, it may cause the patched text to become erroneous. Indeed, it may even cause the original text to be correct. None of the proponents of backtracking have suggested a satisfactory solution to this problem.

6.2 Suppressing Default Reductions

Default reductions are a commonly used space saving technique for LR parsers. Suppose M is an LR parser, and f is its action function. Default reductions are equivalent to creating a new parser M' whose action function f' can be encoded in less space. M and M' are identical except for their action functions. The function f' differs from f only in that for some arguments for which f returns the **error** action, f' returns a **reduce** action. If q is a state such that for some lookahead string x $f(q, x)$ is a **reduce** action, then for every lookahead string y such that $f(q, y)$ is the **error** action, $f'(q, y)$ is a **reduce** action. The language recognized by M' is the same as the language recognized by M.

The only effect default reductions have on parsing is to delay syntactic error detection until after some erroneous reductions have been done. Consider the erroneous statement

$$k := m, + 1$$

where k and m are integer variables. The apparent error is that an extra comma has been inserted. A canonical LR(1) parser for Pascal will not do any reductions involving m before the error is detected. However, an LR(1) parser that uses default reductions will reduce the text preceding the comma to a statement before detecting the error. Thus, if the error recovery system is unable to do backtracking, it will not be able to recover gracefully from this error.

Tests based on the Ripley-Druseikis suite [RD78] indicate that most erroneous reductions are the result of default reductions. Therefore, eliminating default reductions will prevent most erroneous reductions. A side benefit of not using default reductions is that parsers that do not use default reductions can be made to run faster than those that do. Unfortunately, eliminating default reductions can cause the parse tables to grow dramatically. The size of the increase depends on the table packing algorithm used to encode the parse tables. For the parser generator Bison, the size of the parse tables for a grammar for Fortran 77 [ANS78] more than doubled when default reductions were not used. The size of the parse tables for a Pascal grammar were about 2.8 times as large.

A parser that uses default reductions can be extended so that its error checking capabilities equal those of parsers that do not use default reductions. A bit array, called the *default array*, can be used to indicate those contexts in which a default action should be done. The default array is indexed by states and lookahead strings. If the action for a given state and lookahead string is the default action for that state, the entry for that state and symbol is 1; otherwise, it is 0. A parser can avoid performing erroneous default reductions by checking the default array before performing each default reduction to see if the lookahead is legal.

The advantage of using a default array over eliminating default reductions is that less space is needed. The default array for the Pascal grammar mentioned above has been generated. When represented as a simple two-dimensional bit array, the default array occupied 2,648 bytes. About 40% of the rows of the default array contained all zeros, which suggests that standard bit table packing techniques could dramatically reduce the space needed to represent the default array. Except for the default array itself, the tables used by the version of the parser that uses the default array to check the legality of default reductions are identical to those used by the version of the parser that uses default reductions without checking that the lookahead token is legal. Since the default array is about 52% as large as the other tables taken together, the parse tables for the version of the parser that uses the default array are about 52% larger than the parse tables for the version of the parser that uses default reductions without checking their legality. This increase is admittedly large. Nonetheless, the tables for the version of the parser that uses the default array are only about 57% as large as the tables for the version of the parser that does not use default reductions.

It may seem that using a default array would make a parser slower. However, timings have shown that using a default array can make a parser faster. The two parsers used for the timings were versions of the parser used in the Pascal auditor with all semantic operations removed. The only difference between them was that one tested if a default reduction should be done before testing if a shift should be done and the other did not. The version of the parser that did the early test for default reductions was as fast as or slightly faster than the control version for all programs used in the timings. The reason for this result is now clear. Parsers typically perform many more reductions than shifts, and most of those reductions are default reductions. Normally, a parser must check if a shift or a nondefault reduction should be done before applying a default reduction. However, if a default array is used, it is possible to tell if a default reduction should be done without first having to check if other parsing actions should be done. Therefore, the net parsing time is reduced.

Whether suppressing default reductions prevents enough erroneous reductions to permit good error recovery depends on the language being parsed and on the specific parsing technique. For Pascal, suppressing default reductions is not always sufficient, as is shown by the if-statement example given at the start of this chapter. Also, for parsers using SLR(1) tables, just eliminating default reductions will still allow many harmful erroneous reductions. For parsers using LALR(1) tables, suppressing default reductions still allows some erroneous reductions that would be avoided if full LR(1) error checking were performed. However, the Ripley-Druseikis test suite contains no examples of errors where LALR(1) error checking allows erroneous reductions that would not also be allowed with full LR(1) error checking.

6.3 Pretesting

A canonical LR(1) parser never does a reduction if the symbols represented by the contents of the parse stack together with the lookahead symbol do not constitute a correct prefix. Parsers using SLR(1) or LALR(1) tables, on the other hand, sometimes do reductions when the lookahead is not part of a legal input. However, they will never shift over a symbol that is not part of a correct prefix. That fact can be exploited to allow full LR(1) error checking when parsing with SLR(1) or LALR(1) tables.

LR(1) pretesting of a configuration of an LR parser consists of testing if the parser will enter an error configuration after doing zero or more reductions. Pretesting can be implemented by emulating parsing on an auxiliary parse stack. A more efficient implementation is described below. Pretesting can provide full LR(1) error checking in a parser using SLR(1) or LALR(1) tables. Whenever the parser is about to do a reduction while in a state that was entered by shifting over a terminal symbol, it should pretest its current configuration. If pretesting reveals that the parser will enter an error configuration after performing some reductions, the lookahead symbol cannot be part of a legal sentence. In such a case, the error recovery system should be invoked before any reductions are done.

LR(1) pretesting can be implemented by the function *Shiftable* shown in Figure 6.1. *Shiftable* takes a symbol as its argument. For pretesting, the symbol should be the lookahead symbol. *ParseStack* is the global parse stack. *ShadowStack* is a local variable used to record the effects of shifts done during testing. *ShadowStack* is needed because the parse stack must be left unchanged. The variable j indicates the current number of entries in the shadow stack. Note that j is never greater than one unless a λ-reduction is done. If the parser never does a λ-reduction, *Shiftable* can be made much simpler. *Shiftable* also uses some global functions. The function f is the parsing automaton's action function, and g is its goto function. The function *rhslen* takes a rule as its argument and returns the length of its rhs. The function *lhs* takes a rule as its argument and returns its lhs.

Using *Shiftable* to perform LR(1) pretesting amounts to parsing the input text twice. Some of the duplicated effort can be avoided by recording the sequence of rules used in reductions during pretesting. Whenever pretesting reveals that the next token can be shifted, the parser can apply the recorded sequence of rules without having to recompute it.

LR(1) pretesting permits full LR(1) error checking to be done by a parser using SLR(1) or LALR(1) tables. In that respect, pretesting is superior to suppressing default reductions. However, unlike suppressing default reductions, pretesting significantly slows the parser. Timings indicate that LR(1) pretesting may add as much as 15% to the time spent in parsing a program. However, this increase is not as bad as it might seems since the total time a compiler spends parsing is typically less than 5% of the total compilation time. Therefore, pretesting should add less than 1% to the total compilation time.

```
function Shiftable(s: symbol): Boolean;
var ShadowStack: stack of symbol;
begin
    $i \leftarrow$ the index of the top of the parse stack;
    $j \leftarrow 0$;
    $k \leftarrow$ the current state;
    while $f(k, s) =$ reduce $p$ for some rule $p$ do
    begin
        $j \leftarrow j$ - rhslen($p$);
        if $j \leq 0$ then
        begin
            $i \leftarrow i + j$;
            $k \leftarrow g(ParseStack[i], lhs(p))$;
            $j \leftarrow 1$
        end
        else
        begin
            $k \leftarrow g(ShadowStack[j] \leftarrow lhs(p)$;
            $j \leftarrow j + 1$;
        end;
        $ShadowStack[j] = k$
    end;

    if $f(k, s) =$ error then
        return false
    else
        return true
end
```

Figure 6.1 The function *Shiftable*

6.4 LR(k) Error Checking via Stack Restoration

It is possible to provide full LR(k) error checking, for fixed k, in an LR(1) parser using LR(1), SLR(1), or LALR(1) tables. A practical method for implementing LR(k) error checking was suggested by Burke and Fisher [BF82]. While parsing, the numbers of the rules used in reductions are stored in a FIFO queue called the *rule number queue*. Whenever the parser shifts over a terminal symbol, a marker is placed in the rule number queue. The marker indicates which symbol is shifted and the location of any associated semantic data. The parser counts the number of markers in the rule number queue. Whenever there are k markers in the queue, the parser removes entries from it one at a time. Each time a rule number is removed from the queue, the semantic action associated with that rule is applied to the semantic stack. Then, a number of elements equal to the length of the rhs of the rule are popped off the stack and the semantic value associated with the lhs of the rule is pushed onto it. When a marker is removed from the queue, the semantic data associated with the marker is pushed onto the semantic stack, and then parsing resumes. Thus, the Burke-Fisher technique for LR(k) error checking imposes a k token delay between the point when a rule is used during parsing and the point when associated semantic action is applied. Therefore, the parser stack and the semantic stack must be implemented as separate data structures.

After an error is detected, the contents of the parse stack must be restored. Let ℓ be the number of markers in the rule number queue. Then the contents of the semantic stack correspond to the contents of the parse stack before it shifted over the last ℓ tokens. Because of that delay, the semantic stack already contains the proper values. Restoring the parse stack brings it into line with the semantic stack. To facilitate the restoration, Burke and Fisher use yet another stack called the *symbol stack*. The symbol stack is maintained in parallel with the semantic stack. Whenever the parse applies a rule to the semantic stack, it is also applies that rule to the symbol stack. A number of elements equal to the length of the rule's rhs are popped off the symbol stack, and then the symbol on the rule's lhs is pushed onto the symbol stack. Whenever a marker is found, the token named in the marker is pushed onto the symbol stack. Thus, the symbol stack contains the sequence of symbols corresponding to the accessing symbols of the sequence of states in the parser stack just before it shifted over the ℓ-th previous token. Therefore, the parse stack can be restored by successively applying the parser's goto function to the elements of the symbol stack, starting from the parser's initial state. The lookahead can be reconstructed from the markers in the rule number queue.

It may appear that the Burke-Fisher algorithm could be made faster by saving the states corresponding to the earlier version of the parse stack instead of the symbols. The problem is that the states would have to be computed from the goto function. Therefore, the time required to parse error-free text will increase.

There is an alternative way of restoring the parse stack that does not require maintaining a symbol stack. The rule number queue together with a copy of the grammar being parsed provide the information needed to restore the parse stack. The effects of the reductions corresponding to the contents of the rule number queue must be undone in the reverse of the order in which they were done. While there are any rule numbers left in the rule number queue, the effects of the corresponding reductions must be reversed. If the accessing symbol of the topmost state in the parse stack is a terminal symbol, the state is popped off the stack and its accessing symbol is placed in a lookahead buffer. That process is repeated until a state whose accessing symbol is a nonterminal symbol is at the top of the parse stack. That state is popped off the stack. Next, the rule number at the end of the rule number queue is removed from the queue. The states the parser would enter while shifting over the rhs of the indicated rule starting from the current top of the parse stack are then pushed onto the parse stack. This method is used to implement LR(2) error checking in the Pascal auditor.

6.5 Limited Backtracking

The methods described so far for suppressing the effects of erroneous reductions work for syntax errors only. Semantic errors are detected by the semantic routines executed when a reduction is performed. There are cases where a semantic check cannot be done until after some erroneous reductions have been performed. Consider the program fragment shown in Figure 6.2. The likely error is that the comma (',') in the call of *writeln* should have been a period ('.'). The semantic routines are able to determine that *C* cannot be an argument of *writeln* (records cannot be written using *writeln*). However, they cannot do so before *C* has been reduced to the nonterminal for expressions. Since an expression cannot appear as the variable in a selection, the error recovery algorithm will be unable to recover by replacing the comma with a period unless those reductions can be undone.

```
program p(input, output);
type complex =
     record
          re, im: real
     end;
var C: complex;
. . .
begin
     . . .
     writeln(C, re);
     . . .
end.
```

Figure 6.2 A semantic error requiring backtracking

While general backtracking techniques are too slow to be practical, a limited backtracking capability can be provided at a cost comparable to that of the full LR(k) error checking mechanism described in the previous section. The backtracking facility considered here provides the ability to back up both the parser and the semantic analyzer to their configurations just before the parser shifted over the k-th previous token. This facility assumes the model of compilation described in Section 5.3 together with the history based model of symbol tables described in Section 5.4. The method can be extended to handle an inherited attribute stack as well. Therefore, it could be used in an evaluator for LR-attributed grammars.

Suppose LR(k) error checking is to be provided. The limited backtracking facility uses a circular buffer with k+1 entries. Each buffer entry is a record of the form:

```
record
     lowest:        integer;
     top:           integer;
     timestamp:     integer;
     nestinglevel:  integer;
     ParseStack:    array [1 . . MAXDEPTH] of integer;
     SemanticStack: array [1 . . MAXDEPTH] of SemanticType
end
```

where *MAXDEPTH* is the maximum size of the parse stack and *SemanticType* is the type of the entries in the semantic stack. Note that each buffer entry is large enough to contain copies of the entire parse stack and the entire semantic stack. Therefore, k must be small or else too much space will be needed. The parser keeps track of the number of the lowest stack element affected by any reduction since the previous shift. In an implementation for LR-attributed grammars, it must also keep track of the lowest entry in the inherited attribute stack that has been affected. After the parser shifts over a terminal symbol, a snapshot of the entries of the parse stack and the semantic stack that have changed since the previous shift are stored in the buffer. Let ℓ be the number of the lowest element in the parse stack that has changed, and let t be the number of the current top of the parse stack. Then the snapshot consists of an entry in which the field *lowest* is set to ℓ, *top* is set to t, *timestamp* is set to the current value of the timeclock, *nestinglevel* is set to the current nesting level, and elements ℓ through t of the parse stack and the semantic stack are copied into the corresponding locations of the *ParseStack* and *SemanticStack* fields, respectively. Unlike the stack restoration scheme, the limited backtracking facility does not impose a delay between the time a reduction is done and the time the corresponding semantic actions are executed. If there are any free slots in the buffer, the snapshot is stored in the next free slot. If every slot is full, the slot that contains the oldest snapshot is freed and the new snapshot is stored there. The parser and error recovery routines must keep track of the number of full slots.

The buffer provides the information needed to do backtracking. Suppose an error has been detected. Let m be the number of full slots. If the buffer is empty, no backtracking is done. Otherwise, the last $m-1$ tokens to be shifted are placed in a lookahead buffer. Since slots are filled just after doing a shift, the accessing symbol of the state at the top of the recorded segment of the parse stack will be the token the parser shifted to enter that state. Therefore, for each full slot, the token the parser shifted just before the slot was filled can be determined. For each full slot other than the oldest full slot, the token shifted when the slot was filled is determined, and it and its associated semantic value are placed in the lookahead buffer. The symbol table is backed up to the point indicated by the *timestamp* and *nesting level* fields of the oldest full slot in the buffer. The contents of the parse stack and the semantic stack are then restored. The segments of the parse stack and the semantic stack stored in the oldest filled slot are copied back into locations from which they were copied. The top of each stack is reset to value of the *top* field of the oldest slot. Let ℓ be an integer variable. Initially, ℓ is set to the value of the *lowest* field of the oldest slot. Starting from the next oldest slot and working up to the most recently filled slot, the backtracking algorithm checks if ℓ' is less than ℓ, where ℓ' is the value of the *lowest* field of the slot. If it is, entries ℓ' through ℓ of the *ParseStack* and *SemanticStack* fields of the slot are copied back into the corresponding locations of the parse stack and the semantic stack, respectively, and then ℓ is reset to ℓ'.

For correct programs, the efficiency of this backtracking scheme should be comparable to the LR(k) error checking techniques described in the previous section. The scheme's efficiency depends on the size of the entries in the semantic stack. If those entries are large, they will take a long time to copy. However, if they are too large, parsing will be slow regardless of whether any backtracking capabilities are provided. Therefore, the compiler writer already has reasons to make those entries as small as possible.

6.6 Comparing the Techniques

A variety of schemes for avoiding the harmful effects of erroneous reductions have been presented. Each technique other than general backtracking is suitable for use in a practical compiler. Each technique other than general backtracking has been implemented and tested using the Ripley-Druseikis suite of erroneous Pascal programs [RD78]. The Ripley-Druseikis suite consists of 126 Pascal programs containing about 200 errors. The results of those tests are considered in this section.

The Ripley-Druseikis suite revealed some interesting facts about the benefits of suppressing erroneous reductions. The Pascal auditor implements semantic checking as described in Section 4.3 and LR(2) error checking as described in Section 6.4. Disabling both semantic checking and the LR(2) error checking mechanism caused the error recovery algorithm to yield poorer recoveries for 36 errors. Exactly one error was diagnosed more accurately. When only the LR(2) error checking mechanism was disabled, only 7 errors were not handled as well as when both error checks were made. A reason for the improvement is illustrated by the example given at the start of this chapter. In that example, the symbol x in the statement

$$x = 0.0$$

is reduced to a procedure statement and then to a statement before the error is detected. However, if x is not a procedure identifier, the semantic error is detected before x is used in a reduction. Therefore, the error recovery routine is able to find the best repair. When only semantic error checking was disabled, only 7 errors were not handled as well as when both checks were made. Almost every error that was less accurately diagnosed when only semantic checking was disabled was either a procedure call with square brackets around the parameter list or an array reference with parentheses around the subscripts.

There was little difference in the results obtained for the various techniques for avoiding the effects of erroneous reductions. The tests were all done with semantic checking disabled. There was no difference between the results obtained by suppressing default reductions and those obtained by doing full LR(1) error testing. There were two errors for which the recoveries done when using LR(2) error checking were better than those done when using LR(1) error checking. One of those two errors would have been handled just as well in the LR(1) case if semantic checking had been enabled. Frankly, these results are counter-intuitive. It is easy to construct plausible examples for which LR(1) checking is not sufficient. It is surprising that so few such examples are included in the test suite.

The Ripley-Druseikis sample of erroneous programs contains no examples of errors for which limited backtracking proves superior to stack restoration. However, this absence appears to be an artifact of the sample. The sample was created to test syntactic error recovery techniques. Therefore, every error represented in the sample can be detected syntactically. However, an admittedly small sample of erroneous programs gathered locally indicates that limited backtracking can lead to better recoveries for about half of those syntactic errors that are detected semantically.

The test results indicate that checking one symbol of lookahead should prove sufficient for Pascal. Any of the techniques considered could be used to provide that degree of checking. Suppressing default reductions and using limited backtracking would appear to be the best methods. Suppressing default reductions would be the clear choice if speed were the critical factor or if semantic checking were not done. If semantic checking is done, the best recoveries will be produced with limited backtracking.

If LR(k) error checking is to be done, either stack restoration or limited backtracking should be used. If semantic checking is not done during normal compilation, stack restoration would be the method of choice. The semantic restrictions that must be observed to use the limited backtracking techniques are too burdensome to be used simply to gain the advantages of LR(2) error checking. If semantic error checking is done, limited backtracking would be the better choice.

7
Panic Mode for LR Parsers

Panic mode has been used since the late 1950's. Most programming languages then were line-oriented; *i.e.*, each line of a program could be parsed in isolation. Therefore, a compiler recover could from a syntax error by discarding the line in which the error was detected and resuming normal compilation with the following line.

That form of panic mode cannot be used with modern programming languages and compilers. Most modern programming languages treat line ends as ordinary separators, undistinguished from blanks, tabs, and other separators. Furthermore, modern parsing techniques, such as LL(1) and LR(1), are dependent upon left context information that is stored in the parse stack. If a panic mode recovery leaves the parse stack in an improper configuration, the parser will probably detect spurious errors later. Therefore, a panic mode algorithm that simply discards lines will often produce bad recoveries.

Many panic mode algorithms are for use with top-down parsers only. The algorithms used with LR parsers generally do not work as well as those for top-down parsers. Some reasons it is easier to implement panic mode for top-down parsers are discussed in later sections.

This chapter begins with a list of desirable properties for panic mode algorithms. It surveys some existing panic mode algorithms, and examines their strengths and weaknesses. A new panic mode technique for LR parsers that is a synthesis of the best features of some existing algorithms is presented. A user's view of the new technique is given first, followed by a discussion of implementation issues. The chapter concludes with a discussion of how semantic analysis may be affected by panic mode recoveries.

7.1 Desirable Characteristics for Panic Mode Algorithms

This section presents a list of desirable properties for panic mode algorithms. The algorithms presented in later sections are judged by how well they satisfy these properties. The list is, at least in part, subjective.

A panic mode algorithm should recover as soon as possible. That is, as little of the input text as possible should be skipped. The types of errors that will cause an error recovery system to resort to panic mode are, naturally, difficult errors from which to recover. However, if the panic mode algorithm does not recover as quickly as possible, there is a chance that the compiler will fail to detect other errors in the input text. For example, consider the code fragment

```
i := i - 1;  pop the stack }
if i < 0 tehn error;
```

There are two apparent errors here; a left comment brace has been omitted on the first line, and the keyword **then** has been misspelled on the second. Many panic mode algorithms would recover from the first error by skipping ahead to the semicolon on the following line. Therefore, the misspelled keyword on the second line would not be detected.

A panic mode algorithm should not cause spurious errors. This property is both clearly desirable and generally unachievable. It also conflicts with the previous property. If the algorithm tries to recover too soon, there will be instances in which spurious errors will be indicated. For example, consider the code fragment

```
while x > 0 do
      + 1)) + a[k];
```

A possible cause for the error is this example is that a line that should appear between the two lines shown has accidently been deleted. A panic mode algorithm might recover from this error by deleting the text between the keyword **do** and the identifier *a*, and resuming normal parsing with *a*. The array reference *a*[*i*] would then appear to be the start of an assignment statement. However, upon reaching the semicolon, a spurious syntax error will be detected.

A panic mode algorithm should issue informative error messages. Many implementations of panic mode produce vague messages such as "syntax error" or "unexpected input." One reason that programmers make errors that cause panic mode to be invoked is that they do not understand where it is legal to use particular constructs. Therefore, if a recovery is not accompanied by a message indicating what type of construct the panic mode algorithm considered the erroneous construct to be, the programmer is apt to be more confused than aided by the resulting error message. Consider, for example, the program fragment

```
program p(input, output);
begin
    var x, y: real;
    readln(x, y);
```

The apparent error here is that a declaration has been include inside a compound statement. It may be that the programmer believes that declarations should appear inside compound statements. In that case, an uninformative message such as "unexpected input" is apt to be confusing. If, on the other hand, the error message indicated that a statement was expected, the programmer would at least be informed that a statement rather than a declaration was expected in that context.

A panic mode algorithm should be easy for a compiler writer to use. No matter how good an error recovery algorithm may be, if it is hard to use, it will not be used. Some implementations of panic mode require no data other than the parser's tables. Those algorithms require no effort on the part of the compiler writer. However, such algorithms cannot be tuned to take advantage of the compiler writer's knowledge of the language to be compiled. As a result, the recoveries are sometimes of poor quality. At the other extreme, it has been suggested that panic mode could be implemented by having the compiler writer supply a recovery routine for each error action in the parse table. While that scheme could produce good recoveries, the amount of work required of the compiler writer is staggering.

A panic mode algorithm should not be too slow. If an error recovery system includes a good local recovery algorithm, the speed of its panic mode algorithm should not be critical. A study of the Ripley-Druseikis sample of erroneous programs suggests that a good local recovery algorithm should be able to handle at least 80% of the errors that are detected. Therefore, only about one error is five will cause the panic mode algorithm to be invoked. Nonetheless, the panic mode algorithm must not be so slow that when it is invoked, it dominates the total compile time.

7.2 Some Earlier Panic Mode Algorithms

7.2.1 Aho and Ullman's algorithm

Aho and Ullman [AU77] give a simple algorithm for implementing panic mode for LL parsers. The compiler writer supplies the algorithm with a list of *synchronizing tokens*. When the panic mode algorithm is invoked, it skips over the input until it finds a synchronizing token. It then tests if that token can follow the symbol at the top of the parse stack. If so, normal parsing is resumed. Otherwise, the algorithm pops the top symbol off the stack and then loops back to the test. If the parse stack is emptied, compilation is terminated.

The recoveries produced by Aho and Ullman's algorithm is fair at best. Its major flaw is that the set of synchronizing tokens is the same for all contexts. For real programming languages, there is no set of synchronizing tokens that is suitable for every context. For example, consider the case of Pascal. If the token **end** is not a synchronizing token, the algorithm will skip over instances of **end**. Skipping over instances of **end** within the statement part of the program will almost always lead to spurious errors. Suppose, on the other hand, the keyword **end** is always considered to be a synchronizing token. Then if the programmer has erroneously included an instance of the keyword **end** in the declaration part of a block, the panic mode algorithm might recover by terminating the enclosing block.

7.2.2 Pai and Kieburtz' algorithm

Pai and Kieburtz [PK80] present a panic mode algorithm that is essentially an enhancement of the algorihm given by Aho and Ullman. Pai and Kieburtz call synchronizing tokens *fiducial symbols*. Their algorithm uses a different test to decide if the symbol currently at the top of the parse stack can be followed by a particular fiducial symbol. The test succeeds if there is a string whose length is less than or equal to some predefined bound such that the concatenation of the symbols in the parse stack, the string, and the fiducial symbol form a correct prefix. If the test succeeds, the shortest such string is inserted ahead of the fiducial symbol before normal parsing is resumed.

The Pai-Kieburtz algorithm should usually outperform the simpler algorithm presented by Aho and Ullman. For example, when the simpler algorithm is used for Pascal, it does not help to make keywords that start statements, such as **for**, **while**, and **if**, fiducial symbols since, for normal Pascal grammars, they cannot directly follow any symbol that can be pushed onto the parse stack. With the Pai-Kieburtz algorithm, those symbols can profitably be made fiducial symbols. If the algorithm finds one of those symbols while scanning through the input text, it can insert a semicolon ahead of the symbol before resuming normal parsing. Therefore, the algorithm will not have to skip as much of the input text in some cases. However, like the Aho-Ullman algorithm, the Pai-Kieburtz algorithm uses a single set of synchronizing tokens in all contexts. Hence, the Pai-Kieburtz algorithm also suffers from the problem regarding the choice of synchronizing tokens described at the end of Section 7.2.1. Thus, the Pai-Kieburtz algorithm is better than the simpler algorithm presented by Aho and Ullman, but the improvement is small.

7.2.3 Hartmann's algorithm

Hartmann's compiler for Concurrent Pascal [Har77] contains an interesting panic mode algorithm. Hartmann's algorithm is a refinement of the panic mode algorithm used by Ammann [Amm81] in the Zurich implementation of Pascal. The Concurrent Pascal compiler uses a recursive descent parser. Whenever a syntax error is detected, the routine *error* is called. *Error* takes two parameters: the number of the error that was detected, and a set of tokens. The tokens in that set are called the ***key tokens***; they serve the same function as synchronizing tokens. *Error* first prints an error message corresponding to the error number. Next, it skips through the input text until it either finds a key token or reaches the end of file. Normal parsing then resumes.

The set of key tokens passed to *error* depends on context. The parse routines for each language construct, such as statements, declarations, and expressions, take a set of tokens as a parameter. That set is to be the set of key tokens for that construct. For example, when the routine for parsing if-statements calls the routine for parsing expressions, the argument of the call will be the set of key tokens for the if-statement plus the token **then**. The set of key tokens for a construct is the set of all tokens that are permitted to follow that construct.

Hartmann's algorithm works remarkably well. Unlike the previous algorithms, it recovers quickly from most errors and rarely creates situations in which spurious errors will be detected. The method is not flawless, but its flaws are minor. There are some cases where the Pai-Kieburtz algorithm outperforms Hartmann's algorithm. Those cases arise because the Pai-Kieburtz algorithm is able to insert as well as delete when recovering from an error. For example, if the keyword **begin** is missing from the start of a block in a Pascal program, Hartmann's algorithm will consider the remaining input text to be part of the declaration part of the program (until it reaches an occurrence of **begin**). The Pai-Kieburtz algorithm will scan through the input text until it finds a keyword that starts a statement. It will then recover by inserting the token **begin** ahead of the keyword. Hartmann's algorithm is also less "automatic" than most other algorithms. The compiler writer must write all the calls to *error* himself. He must also maintain the correct values for the sets of key tokens.

7.2.4 The Yacc algorithm

The algorithm used in Yacc parsers [Joh78] is perhaps the best-known panic mode algorithm for LR parsers. The Yacc algorithm makes use of a special token called the ***error token***. The error token is used to indicate those contexts from which parsing may continue after an error. The error token may appear as part of the rhs of any rule. When an error is detected, the panic mode algorithm prints a message and checks if the topmost state of the parse stack permits a shift over the error token. If not, states are popped off the parse stack until a state that does permit a shift over the error token is found. The parser then performs a shift over the error token and attempts to parse the remaining input. However, if another error is detected before the parser shifts over three input tokens, the panic mode algorithm is reinvoked, but the error message is suppressed.

The Yacc algorithm rates badly by most of the standards given in the previous section. It not only fails to recover quickly, it frequently fails to recover at all. For example, suppose that the grammar includes the productions

statement → **if** expression **then** statement

expression → **error**

where **error** is the error token. Suppose that the input to the parser includes an if-statement in which the token **then** is misspelled. Further, suppose the parse detects an error within the expression part of the if-statement that causes panic mode to be invoked. Then the panic mode algorithm will pop the stack back to the state entered when it shifted over the keyword **if**. The only token that can be shifted over after shifting over **error** from that state is **then**. However, since the following instance of **then** is misspelled, it will not be recognized. If there are no later instances of the token **then** in the program, the algorithm will skip over the rest of the input text. The algorithm also leads to many spurious errors. If the program mentioned in the previous example contained another if-statement in a later procedure declaration, the panic mode routine would skip to the instance of the token **then** in that procedure and resume parsing. As a result, many spurious errors are likely to be detected. The standard error message "syntax error" is not much aid to the user. Also, it is not easy for a compiler writer to use the technique. Adding productions containing the error token to a grammar will often cause it to cease to be LALR(1), forcing the compiler writer to modify his grammar. However, the technique is reasonably efficient.

The panic mode algorithm used in the Pascal auditor is a modified version of the algorithm used by Yacc. Much of the trouble with the Yacc algorithm stems from the fact that parsing must continue starting from the rightmost state in the parse stack that contains a shift over **error**. The modified algorithm tries to match the remaining input with any state in the parse stack that contains a shift over **error**. The algorithm starts by determining the set of synchronizing tokens. For each state in the parse stack that contains a shift over **error**, the algorithm determines the set of symbols that the parser could shift over after shifting over **error**. The set of synchronizing tokens is the union of those sets. The algorithm skips through the remaining input text until it finds a synchronizing token. It then pops the parse stack back to the rightmost state that can shift over that token after shifting over **error**. Finally, it shifts over **error** and normal parsing resumes. The modified algorithm does not include a call to a standard error reporting routine. The action routines associated with the rules that permit shifts over the error token must generate appropriate error messages.

The modified Yacc algorithm used by the Pascal auditor corrects many of the faults of the original. The modified algorithm recovers from most errors and usually recovers as quickly as possible. Spurious errors resulting from bad recoveries are rare. If the compiler writer is careful, the error messages can be informative aids to the users. However, in one respect the modified algorithm is worse than the original. The quality of the recoveries performed by the modified algorithm depends in part on the exact form of the grammar from which the parser is generated., The forms that lead to good results from the panic mode algorithm are sometimes not ones that would normally be used. One such example occurs in the grammar for the Pascal auditor shown in Appendix A. To avoid some bad panic mode recoveries, the rules for declaration sections were factored one level more than would otherwise have been necessary. Before the grammar was changed, whenever a declaration appeared in the statement part of a block, the panic mode algorithm would discard the portion of the parse stack above the state for the declaration section of the block, issue a message stating that the text between the declaration section and the erroneous declaration constituted a malformed declaration, and then continue parsing as if the statement part of the block had not yet started.

7.2.5 Burke and Fisher's algorithm

Burke and Fisher [BF82] proposed a panic mode technique that works for both LL and LR parsers. They require the compiler writer to specify which tokens are the brackets of the major bracketed structures. The opening brackets are called *scope openers*, and the closing brackets are called *scope closers*. The term scope is used in a syntactic sense; it does not refer to the scope of names. The panic mode technique recovers by discarding portions of the parse stack and the remaining input, and by inserting scope closers. When an error is detected, the panic mode algorithm tests if parsing could continue with the remaining input if a portion of the parse stack were discarded and zero or more scope closers were inserted ahead of the remaining text. The parse must be able to shift over a predefined number of tokens (five in their implementation) for the test to succeed. The portion of the stack to be discarded must all be to the right of the rightmost entry inserted when the parser shifted over a scope opener. If the test succeeds, the stack and the input are modified as indicated, and normal parsing resumes. Otherwise, the first symbol in the remaining text is deleted and the algorithm begins again. The algorithm goes on until either a recovery is found or the input is exhausted. When a recovery is found, the algorithm tries to determine the type of syntactic construct expected in the context where the error was detected. If it can identify the expected construct, the error message it produces will name that construct. Otherwise, a general error message is issued.

The Burke-Fisher technique works well in most cases. Because their algorithm does not permit an arbitrary portion of the parse stack to be deleted, there are some errors from which the algorithm cannot recover. Also, because of the long parse check, there is a chance that the algorithm will ignore the best recovery because of a later unrelated error. However, the types of errors that cause those problems are rare. A more common problem of their method is that the text that is deleted often does not correspond to any natural unit of the program. Therefore, it is sometimes difficult to determine what the actual error was. Also, their recoveries can turn one type of statement into another. For example, if there is an error in the expression on the right-hand side of an assignment, their algorithm will sometimes delete all symbols in the statement other than the variable on the left-hand side of the assignment, thus turning the assignment statement into a parameterless procedure call. Their technique does have the advantage of being easy to apply. A compiler writer need only supply the lists of scope openers and scope closers.

7.2.6 Sippu and Soisalon-Soininen's algorithm

The final algorithm considered here was proposed by Sippu and Soisalon-Soininen [SS83]. Their algorithm was implemented as part of an LALR(1) parser generator. The algorithm recovers from an error by substituting a single nonterminal symbol for portions of the parse stack and the remaining input text. The nonterminal symbol is called a *reduction goal*. For example, if the algorithm finds that the states at the top of the stack and the start of the remaining input seem to constitute a malformed statement, those parts of the stack and the input are deleted, and the algorithm then performs a shift over the nonterminal symbol for a statement.

Sippu and Soisalon-Soininen introduce the notion of feasible reduction goals. Each nonterminal symbol such that there is a state in the parse stack that permits a shift over that symbol is a potential reduction goal. Let $q_1 \ldots q_k q_{k+1} \ldots q_n$ be the contents of the parse stack, and let A be a nonterminal symbol such that q_k includes a shift over A. Let x be the string consisting of the accessing symbols of $q_{k+1} \ldots q_n$ taken in sequence. A is a *feasible* reduction goal if and only if x is the prefix of a string y such that $A \overset{*}{\underset{rm}{\Rightarrow}} y$. Only feasible reduction goals are to be considered when selecting a recovery.

Sippu and Soisalon-Soininen did not test for feasible reduction goals in their implementation. Instead, they use a test for a weaker condition they call *weak feasibility*. A reduction goal A is weakly feasible if the accessing symbol of the following state on the parse stack is the first symbol of some string that can be derived from A. They do not explain why they chose to implement this weaker test. The reason may have been speed since the algorithm they give for testing feasibility is not very efficient.

The panic mode algorithm described by Sippu and Soisalon-Soininen tries to find the token in the remaining input that matches a feasible reduction in the parse stack such that the sum of distance of the token from the start of the remaining input and the distance of the reduction goal from the top of the parse stack is minimized. The algorithm alternates between testing new tokens against the reduction goals near the top of the parse stack and testing tokens near the start of the remaining input against reduction goals further down in the parse stack. Once a match that allows parsing to continue is found, the input preceding the token that was matched is discarded, and the contents of the parse stack above the reduction goal that was matched are popped.

The algorithm by Sippu and Soisalon-Soininen should usually produce good recoveries. However, the error messages produced by their implementation are apt to be confusing. Each error message indicates which symbols were deleted and which nonterminal was substituted for them. Grammars used to implement compilers are not usually designed to be readable. Therefore, there is a good chance that an error message will refer to a nonterminal symbol that will make no sense to a naive user.

7.2.7 Properties that lead to good panic mode recoveries

A number of properties that affect the quality of panic mode recoveries have been revealed by this survey. It has shown that

1. the choice of recoveries should be determined by the context in which the error was detected,
2. recoveries should be allowed to insert tokens as well as delete them,
3. the compiler writer should be able to determine which symbols can be involved in a recovery, and
4. the compiler writer should not be required to add special rules to his grammar to support panic mode.

Most programming languages divide programs into sections that possess different syntactic structures. As a result, those algorithms that did not take context into account performed poorly for at least some contexts. Those recovery algorithms that performed insertions were often able to recover sooner than those that did not. The recovery algorithms that do not place restrictions on which symbols can be involved in recoveries sometimes produce confusing recoveries. Finally, those algorithms that require the grammars to be modified are harder to use than those that do not. Changes to a grammar may change its grammar class, thus making it unacceptable to the parser generator. Further, some constructs must be described using nonintuitive rules to make the algorithm produce desirable recoveries.

Study of the differences between the panic mode algorithms used with top-down parsers and those used with LR parsers reveals one reason why it is easier to produce good panic mode algorithms for the top-down parsers. The parse stack of a top-down parser contains all of the nonterminal symbols that the parser is trying to reduce. A recovery by a panic mode algorithm consists, in part, of determining which of those symbols should be completed. For LR parsers, the parse stack does not immediately reveal which nonterminals are to be reduced; indeed, it is in general impossible to determine that information. However, Sippu and Soisalon-Soininen's concept of feasible reduction goals provides a close approximation to that information.

7.3 Panic Declarations

This section gives a compiler writer's view of a new technique for implementing panic mode. The new technique is similar to that of Sippu and Soisalon-Soininen, but it gives the compiler writer greater control over the selection of recoveries. It is not an "automatic" algorithm in that the compiler writer must supply some additional information to the parser generator. The extra information consists of directives called *panic declarations*. The syntax for panic declarations given below should be considered illustrative, not definitive.

The simplest form of a panic declaration is

%**panic** *nonterminal string*

where *nonterminal* is a nonterminal symbol and *string* is a string delimited by double quotes. The nonterminal symbol must not be left recursive for reasons given later. The declaration indicates that the specified nonterminal symbol is an acceptable reduction goal for the panic mode algorithm. The string is a print name for the nonterminal symbol to be used in error messages. The new panic mode algorithm would produce essentially the same recoveries as those now produced by the Pascal auditor if the following panic declarations were added to the Pascal grammar:

```
%panic   pgmhead   "program header"
%panic   prchead   "procedure header"
%panic   fnchead   "function header"
%panic   parpack   "parameter list"
%panic   dcl       "declaration"
%panic   stmt      "statement"
%panic   expr      "expression"
```

More specific panic declarations could be given, which would allow the panic mode algorithm to produce more specific error messages. For example, panic declarations could be given for each kind of statement in addition to the declaration for general statements. Adding rules containing error tokens to achieve the same purpose for a Yacc-like panic mode algorithm would probably introduce LALR(1) conflicts.

If every panic declaration is of the form described above, the parser generator will compute the set S of tokens that can follow the listed nonterminal symbols and a finite state machine M, which will be used to check for feasibility. The members of S are *fiducial symbols*. When the panic mode algorithm is invoked, it will skip through the input text until it finds an occurrence of a fiducial symbol. It then scans through the parse stack, going from right to left, looking for a state that permits a shift over a nonterminal symbol that is a named in a panic declaration. Each time it finds such a state, it tests if any of the nonterminal symbols named in panic declarations that can be shifted from that state are feasible reduction goals. If that test succeeds, the algorithm then tests if the parser could shift over the lookahead symbol after shifting over any of the feasible reduction goals. If a state passes both tests, the panic mode algorithm will pop all states above that state, shift over one of the feasible reduction goals, issue an error message (if the panic declaration for the goal symbol included a string), and resume normal parsing. If no state passes the tests, the current fiducial symbol is discarded, and the algorithm goes back to skipping over the text. If the input is exhausted before a suitable recovery is found, the algorithm will issue a message stating that it was unable to recover and compilation will be halted.

The new panic mode algorithm issues two forms of error messages. One form indicates that the goal symbol is missing. That form is used if the recovery does not involve discarding a portion of the parse stack or input text. That type of recovery is equivalent to inserting the goal symbol at the error's detection point. The other form states that the goal symbol is malformed. That form is used whenever the goal symbol replaces one or more symbols in the parse stack or input text.

The productions that define the nonterminal symbols named in panic declarations affect the efficiency and the quality of the recoveries performed by the new panic mode algorithm. For example, suppose the symbol *expr* is the nonterminal symbol for expressions, and suppose that *expr* is named in a panic declaration. In a normal grammar for expressions, *expr* would appear on the rhs of the rule defining parenthesized expressions. Now, consider the code fragment

$$i \ := \ (((((i;$$

In this admittedly unrealistic example, there would be six states on the parse stack that would allow a shift over *expr*, one for each level of parenthesization. Therefore, the panic mode algorithm must check if all six states allow shifts over feasible reduction goals, and since they all do, it must determine which of them permit a shift over a semicolon following a shift over a feasible reduction goal. Only the state corresponding to the symbol ':=' will pass the latter test. The grammar for expressions could be rewritten to avoid the need for a portion of that work. Another symbol, say *expr1*, could be used in the recursive definition of expressions. In particular, the rhs of the rule defining parenthesized expressions would name *expr1* instead of *expr*. The symbol *expr* named in the panic declaration could then be defined as

$$expr \rightarrow expr1$$

Therefore, in the previous example, the only state in the parse stack that would allow a shift over *expr* would be the state corresponding to the symbol ':='.

There must also be a mechanism for specifying semantic values. For example, the panic declaration for expressions might be

```
%panic  expr "expression"
        { $$ = make_error_node(); }
```

where *make_error_node* is a function that returns a special error value.

The panic mode scheme described so far possesses all of the desirable properties listed in the previous section except for the ability to insert tokens. Allowing insertions is sometimes necessary to make any recovery possible. Consider, for example, the program

```
program p(output);
    var i: integer;
begini := 1;
    if i = 1 then  writeln("ok")
end.
```

The apparent error is that the keyword **begin** and the identifier *i* have been run together. The lexical analyzer will recognize the merged tokens as a single identifier. The semicolon at the end of a declaration in Pascal is commonly made a part of the syntax for a declaration (to avoid an LR(1) parsing conflict). Assume that the grammar has been so defined in this case. The set of fiducial symbols for a declaration will be **const**, **type**, **var**, **procedure**, **function**, **begin**, and perhaps **label**. Therefore, if the local recovery algorithm is unable to find a repair, and the panic mode algorithm is not allowed to insert the keyword **begin**, there is no way to recover from the error.

Allowing arbitrary insertions can lead to ridiculous recoveries. Therefore, the compiler writer should be given strict control over the choice of insertions to be allowed during a panic mode recovery. As a general rule, insertions should be allowed only before keywords that can only appear in a single context. For example, in Pascal, it would be reasonable to allow inserting a semicolon before every keyword that could begin a statement, except possibly the keyword **case** which can also appear as part of a variant record.

The compiler writer specifies which insertions are to be allowed by giving a list of pairs of tokens. The first token of the pair is the token that can be inserted. The second token is a fiducial symbol. Whenever the specified fiducial symbol is encountered during a forward scan by the panic mode algorithm, for each instance of the nonterminal symbol specified in the panic declaration that is found to be feasible, the panic mode algorithm will test if it would be possible to shift over both tokens of the pair (in order). If so, the first token is inserted and the indicated recovery is carried out. For example, to handle the error in the previous example, the compiler writer could supply the panic declaration

```
%panic   dcl   "declaration"
               <begin, if>
```

The panic mode algorithm would then be able to recover upon finding the token **if** by inserting the keyword **begin** ahead of it.

Writing all those pairs can become tedious. As a shorthand, a list of symbols can be defined with a list declaration. The form of a list declaration is

%list *name* $token_1 \ldots token_n$

The name of a token list must not be the same as the name of a token. If a list name appears as part of a panic declaration, it is equivalent to all possible combinations of pairs formed from the elements of the list. For example, the declarations

```
%list     stdelim    else semicolon
%list     stkey      begin case for goto if repeat while with

%panic    stmt "statement"
          <stdelim, stkey>
```

specify that the token **else** or the token semicolon can be inserted ahead of any of the keywords that can appear at the start of a statement. The insertions are tested in their order of appearance. Thus, in the previous example, the algorithm would try to insert **else** before it tried to insert a semicolon. A complete set of panic declarations for Pascal is shown in Figure 7.1.

```
%list     stdelim    else semicolon
%list     stkey      begin case for goto if repeat while with


%panic    pgmhead    "program header"

%panic    prchead    "procedure header"
     { new_scope(); }

%panic    fnchead    "function header"
     { new_scope(); }

%panic    parpack    "parameter list"

%panic    dcl        "declaration"
     < begin, stkey >

%panic    stmt       "statement"
     < stdelim, stkey >

%panic    expr       "expression"
     <rpar, semicolon>
     { $$ = make_error_node(); }
```

Figure 7.1 Panic declarations for Pascal

One category of panic mode recoveries is handled in a special way. Consider the code fragment

writeln(*x*, *y*, ((*x* * *y*);

where there are two unmatched left parentheses. Assume that the local recovery algorithm cannot patch the error. If the panic declaration for expressions does not allow any insertions, the likely panic mode recovery for this example would indicate that the entire statement is malformed. A panic declaration such as

```
%panic    expr "expression"
          <rpar, semicolon>
```

where "rpar" is the name for a right parenthesis, would allow the panic mode algorithm to recover by replacing the text starting from the second left parenthesis in the statement and ending with the right parenthesis with the nonterminal symbol *expr*. However, a naive implementation of the panic mode algorithm would produce a pair of error messages: one stating that the text that was replaced constitutes a malformed expression, and another indicating that a right parenthesis was inserted. However, the programmer is unlikely to consider the right parenthesis to be a part of the expression; rather, he will probably consider it to be the match of the left parenthesis following the identifier *writeln*. The error message can be brought into line with the programmer's likely intent at the cost of a few additional tests. If the accessing symbol of the state at the top of the parse stack is the same as the token to be inserted, and if the first token of the remaining input (before discarding any of the input) is the fiducial symbol of the recovery, the insertion is not done. Instead, the top state is popped off the parse stack, and its accessing symbol is attached to the front of the remaining input. The algorithm then recovers normally.

The system described above allows only one token to be inserted as part of a panic mode recovery. It is easy to find examples where inserting more than one symbol would lead to a better recovery. However, it is harder to implement systems that allow more than one token to be inserted.

7.4 The New Panic Mode Algorithm

The test for feasibility is the most complex aspect of the new panic mode scheme. Sippu and Soisalon-Soininen [SS83] give an algorithm for testing feasibility in their article. However, their algorithm seems to be needlessly complex and inefficient. An alternative algorithm is given below. The algorithm is given for LALR(1) parsers, but could be adapted for use with other classes of parsers.

The test for feasibility depends on the following facts. Let $G = (V, \Sigma, P, S)$ be an LALR(1) grammar, and let M be the LALR(1) parser for G. Let $q_1 \ldots q_n$ be any sequence of states that can be the contents of the parse stack during the parse of a string. Let $x = a_1 \ldots a_n$ be the string consisting of the accessing symbols of the q_i's taken in sequence. The string x must be the prefix of a sentential form of G. Let STA be the state transition automaton of M. Since x is the prefix of a sentential form, STA must be able to shift over x without entering an error state. Let A be a nonterminal symbol such that the parser can shift A while in state q_i, $1 \leq i \leq n$. Let $G_A = (V, \Sigma, P, A)$, let M_A be the LR(0) parser for G_A, and let STA_A be the state transition automaton for M_A. Let $z = a_{i+1} \ldots a_n$. Then z is the prefix of a string w such that $A \overset{*}{\underset{rm}{\Rightarrow}} w$ if and only if STA_A can shift over z without entering an error state. Therefore, testing if A is a feasible reduction goal for q_i is equivalent to testing if STA_A can shift over z without entering an error state.

A practical algorithm for testing feasibility can be based on state transition automata. When a parser is constructed, the parser generator also builds the state transition automata for every symbol named in a panic declaration. (Portions of the STAs can obviously be shared.) For each state, the parser generator builds a list of all nonterminal symbols named in panic declarations that the parser can shift while in that state. Whenever the panic mode algorithm is invoked, it constructs the string of accessing symbols of the states in the parse stack. To test if a nonterminal symbol A is a feasible reduction goal starting from the i-th state in the parse stack, the STA for A is run over the string of accessing symbols starting from the $i+1$st symbol. If the STA does not enter an error state, A is a feasible reduction goal.

A simple change to the feasibility algorithm can improve its efficiency. Many states of an LALR(1) parser have only one item in their kernel set. Suppose $A \rightarrow x \, . \, y$ is the only kernel item of a state q. If q is the j-th element of the parse stack, the string $a_{j+1} \cdots a_n$ must be the prefix of a string z such that $y \overset{*}{\underset{rm}{\Rightarrow}} z$. If an STA shifts over the j-th symbol in the string of accessing symbols, it will be able to shift over the remainder of the string. Thus, the test for feasibility automatically succeeds if the j-th symbol is shifted, and so the test can be terminated at that point. Note that the only extra data structure needed to implement this improvement is a Boolean vector that indicates for each state whether the kernel item set of that state contained only one item. Further, if a portion of the state transition automaton for a nonterminal symbol is accessed only through transitions that correspond to states whose kernel item sets contain only one item, that portion of the automaton will no longer be referenced. Therefore, space can be saved by eliminating those parts of the automaton.

A more ambitious modification can reduce the order complexity of testing feasibility. Suppose that the algorithm presented above is used to find every feasible reduction goal for a given configuration of the parse stack. Assume that the parse stack contains n states. There are a bounded number of reduction goals for each state. For each reduction goal, the algorithm's worst-case time complexity is $O(n)$. Therefore, the worst-case complexity is $O(n^2)$. An alternative algorithm has a worst-case upper bound of $O(n)$. Let STA_A be the state transition automaton for the symbol A. Because STA_A is a finite automaton, it is possible to construct another finite automaton RPA_A that recognizes the reverse of the prefixes of the strings accepted by STA_A. The set of feasible reduction goals can be computed as follows. First, construct the string of accessing symbols of the states in the parse stack. Then, for each symbol A named in a panic declaration, run RPA_A over the reverse of that string. Whenever RPA_A enters a final state, check if the state in the parse stack corresponding to the symbol just shifted permits a shift over A. If so, A is a feasible reduction goal for that state. Since there are a bounded number of symbols, and since each RPA requires at most $O(n)$ time to scan the reversed string, the worst-case complexity is $O(n)$. The constant of proportionality can be improved by merging the RPAs for the symbols named in panic declarations into a single finite state machine. It is unclear whether this method of finding feasible reduction goals or the previous method will prove better in practice.

Suppose that the panic mode algorithm has found a fiducial symbol for which there are two or more feasible reduction goals. Each reduction goal is, of course, associated with a position in the parse stack. If one goal is associated with a position to the right of the positions associated with every other feasible reduction goal, that goal is selected as the one to be used in the recovery. If there are two or more feasible reduction goals associated with the rightmost position for which there are any feasible goals, there must be some rule for choosing among them. For example, suppose the input to the parser generator included the panic declarations

```
%panic   stmt     "statement"
%panic   call     "procedure-statement"
%panic   assn     "assignment-statement"
%panic   ifstmt   "if-statement"
```

Then for the code fragment

$$\textbf{if } i < 0 \; tehn \; 2;$$

both *stmt* and *ifstmt* will be feasible reduction goals. To provide the most information to the user, the more specific of the two, namely, *ifstmt* should be selected. On the other hand, for the code fragment

$$p9x, y);$$

stmt, call, and *assn* are all feasible reduction goals. Since the statement could be either a procedure statement, or an assignment statement, the more general *stmt* should be chosen. There are examples where choosing the more specific goal is wrong even when it is the only one of the more specific goals that is feasible. For example, given the code fragment

$$a(i) = 0;$$

the parser will shift over every token up to and including the right parenthesis. Therefore, the only feasible reduction goals will be *stmt* and *call*. In this case, it is clearly better to choose the more general construct *stmt* over the more specific one *call*. This type of situation is rare for Pascal, but is more common for C [KR78].

The examples above suggest the following rules for selecting among feasible reduction goals. Let S be the set of nonterminal symbols named in panic declarations. A partial order relation R can be defined over S. The relation $x\text{R}y$ is true if and only if $x \overset{*}{\underset{rm}{\Rightarrow}} yz$ for some string z. R is well-defined because left-recursive symbols cannot be named in panic declarations. Let F be a set of reduction goals. For each pair of symbols x, y in F, one symbol will be preferred over the other. If $x\text{R}y$ and there is no other symbol z in F such that $x\text{R}z$, then y is preferred over x. If $x\text{R}y$ and there is at least one symbol z in F such that $x\text{R}z$, then x is preferred over y. If no relation is defined between x and y, the symbol named earlier in the list of panic declarations submitted to the parser generator is preferred. Therefore, any set of reduction goals can be sorted by order of preference.

The new panic mode algorithm is given in Figure 7.2. The contents of the parse stack are denoted by $q_1 \ldots q_n$. The set S of fiducial symbols consists of all tokens that can follow a nonterminal symbol named in a panic declaration plus all tokens explicitly named as fiducial symbols in panic declarations. For each state q, the set G_q consists of every nonterminal symbol A named in a panic declaration such that there is a shift from q over A. For each state q and each token t, $I_{q,t}$ is the list of symbols which can be inserted before t as part of a recovery in which q is the reduction goal. The function *feasible* takes a nonterminal symbol, a string, and the index of a position in the parse stack as arguments. It returns true if the symbol is a feasible reduction goal of the indexed position and false otherwise. The function *shiftable* takes a sequence of states and a string as arguments. It returns true if the parser would be able to shift over the

```
procedure $PanicMode$;
begin
    $first \leftarrow true$;
    $recovered \leftarrow false$;
    for $i \leftarrow 1$ to $n$ do
        $a_i \leftarrow$ the accessing symbol of $q_i$;
    while not $recovered$ and the input has not been exhausted do
    begin
        $t \leftarrow$ the next input token;
        advance the input by one token;
        if $t \in S$ then
        begin
            for $i \leftarrow n$ downto 1 do
            begin
                $\ell \leftarrow 0$;
                for $x \in G_{q_i}$ do
                    if $feasible(x, a_1 \ldots a_n, i)$ then
                    begin
                        $\ell \leftarrow \ell + 1$;
                        $g_\ell \leftarrow x$;
                    end;
                $m \leftarrow 0$;
                for $j \leftarrow 1$ to $\ell$ do
                    if $shiftable(q_1 \ldots q_n, g_j t)$ then
                    begin
                        $m \leftarrow m + 1$;
                        $h_m \leftarrow g_j$;
                    end;
                if $m > 0$ then
                begin
                    $A \leftarrow preferred(h, q_i)$;
                    $name \leftarrow$ the print name of $A$;
                    if $first$ and $i = n$ then
                        issue the message 'missing $name$'
                    else
                        issue the message 'malformed $name$';
                    pop all elements above $q_i$ off the parse stack;
                    $DoParse(A)$;
                    $DoParse(t)$;
                    $recovered \leftarrow true$;
                end
                else
                begin
                    $k \leftarrow 1$;
                    while $k \leq |I_{q_i,t}|$ and not $recovered$ do
                    begin
                        $s \leftarrow$ the k-th element of $I_{q_i,t}$;
                        $m \leftarrow 0$;
                        for $j \leftarrow 1$ to $\ell$ do
                            if $shiftable(q_1 \ldots q_n, g_j st)$ then
```

```
                                        begin
                                            $m \leftarrow m + 1$;
                                            $h_m \leftarrow g_j$;
                                        end;
                                    if $m > 0$ then
                                    begin
                                        $A \leftarrow$ preferred($h$, $q_i$);
                                        name ← the print name of $A$;
                                        if first and $i = n$ then
                                            issue the message 'missing name'
                                        else
                                            issue the message 'malformed name';
                                        if not first or $i \neq n - 1$ or $s \neq a_n$ then
                                            issue the message 'inserted $s$';
                                        pop all elements above $q_i$ off the parse stack;
                                        DoParse($A$);
                                        DoParse($s$);
                                        DoParse($t$);
                                        recovered ← true;
                                    end;
                                    $k \leftarrow k + 1$;
                                end
                            end
                        end
                    end;
                    first ← false;
                end;
                if not recovered then
                begin
                    issue a message stating that an unrecoverable error has been detected;
                    terminate compilation;
                end
            end
```

Figure 7.2 The new panic mode algorithm

string (possibly after some reductions) if the parser contained the sequence of states and false otherwise. *Shiftable* is a generalization of the function described in Section 6.3. The function *preferred* takes a sequence of nonterminal symbols and a state as arguments. It returns the symbol of the sequence that is preferred over all the others according to the preferencing scheme described above. The procedure *DoParse* takes a symbol as its argument. If the symbol is a nonterminal symbol, it shifts over the symbol. If the panic declaration for the symbol associates a semantic action with it, that action will be executed. If the symbol is a terminal symbol, *DoParse* will parse until it shifts over the symbol. Semantic actions associated with any reductions that are performed will be executed ignoring the results of semantic checks.

The panic mode algorithm assumes that some technique for deferring reductions is being used. It assumes that the parse is advanced up to the point at which the parser shifts over the rightmost token preceding the detection point of the error. Therefore, unless the first token in the program was the source of the error which caused the algorithm to be invoked, the accessing symbol of the state at the top of the parse stack will be a terminal symbol.

The new panic mode algorithm has been implemented by Michael C. Shebanow, a computer science graduate student at Berkeley. Parsers using the new algorithm have been constructed for C and Modula-2 [Wir83]. Good results were obtained for Modula-2 with little effort. The grammar used to generate the parser was a straightforward adaptation of the grammar presented in [Wir83]. A handful of panic declarations were added to the grammar. The resulting implementation of panic mode produced good recoveries.

The results for C initially were not as good. The problem was that in C, semicolons are used as statement terminators rather than as separators. Therefore, making the nonterminal symbol for statements a reduction goal of a panic declaration caused all symbols that could start a statement to be fiducial symbols. Because C is an expression language, identifiers, constants, and unary operators can all appear at the start of a statement. As a result, the panic mode algorithm often recovered too soon. The problem could have been solved by rewriting the grammar so that semicolons were treated as separators. However, Shebanow solved the problem by allowing the compiler writer to declare that certain symbols should not be considered fiducial. In his implementation, identifiers, constants and unary operators are declared nonfiducial.

Although the new algorithm has not been implemented for Pascal, it has been hand-simulated for those errors in the Ripley-Druseikis suite that caused the Pascal auditor to invoke the panic mode algorithm. The panic declarations shown in Figure 7.1 were used for the simulation. Even with only seven panic declarations, the new algorithm performed at least as well as the Pascal auditor and Berkeley Pascal [GHJ79] in all cases. The new algorithm also produced recoveries as good as or better than the Burke-Fisher techniques in most cases. However, there were a few programs for which the Burke-Fisher system produced better recoveries than the new algorithm. For example, consider the code fragment

```
end; * test *
x := 1
end.
```

The apparent error is that the programmer has used malformed comment brackets. The Burke-Fisher algorithm recovers from this error at the end of the first line, while the new algorithm does not recover until the end of the second line. The Burke-Fisher algorithm was able to recover at the end of the first line because it treats all symbols as fiducial symbols. The new algorithm did not recover at the end of the first line because, for the given set of panic declarations, identifiers are not among the fiducial symbols. However, because the Burke-Fisher system treats all symbols as fiducial symbols, it sometimes recovers too quickly and so detects spurious errors.

7.5 Semantics and Panic Mode

Semantic information is of little use in panic mode recoveries. Each semantic check and each semantic action is tied to a particular class of syntactic objects. In panic mode, no effort is made to identify the syntactic components of the text being scanned. Therefore, semantic information about the text is, for the most part, unavailable.

A panic mode recovery can cause parsing and semantic analysis to become out of step. For example, if a panic mode recovery for a Pascal program causes a with-statement to be terminated, no further syntactic problems would be expected. However, if declarations were entered into the symbol table during semantic analysis of the with-statement and those entries were not removed because an error led to a panic mode recovery, spurious semantic errors might be detected later.

Many compilers do not perform semantic analysis until the entire program has been parsed. In those compilers, syntactic error recovery cannot affect semantic analysis. Further, many compilers do not perform semantic analysis at all if any serious syntactic errors were detected. That approach to the problem is obviously incompatible with the error recovery techniques developed in this work.

The problem of ensuring that semantic analysis does not get off track can be partially solved by an automatic technique. The technique requires the symbol table to be organized as in the limited history scheme described in Section 5.4. Recall that in that scheme, the nesting level that was current at the time a state was pushed onto the parse stack is recorded. Suppose that the panic mode algorithm presented in the previous section is invoked and that it finds a recovery. Let q_i be the state in the parse stack that has been matched with a reduction goal. Let L be the nesting level recorded for q_i. If L is less than the current nesting level, all entries in the symbol table that were entered at a nesting level greater than L should be removed and then the current nesting level should be reset to L.

The technique just described is not always adequate. The syntactic constructs derived from a single nonterminal symbol usually have no net effect on the nesting level. The semantic actions executed while analyzing a construct may increase the nesting level at some times and decrease it at others, but the net effect is to leave it unchanged. For example, in Pascal, a with-statement will increase the nesting level once for each record variable named in its with-list, but at the end of the statement, the nesting level will be the same as it was before the with-statement was encountered. The automatic technique is adequate for panic mode recoveries where the chosen reduction goal has that property. If a reduction goal does not have that property, the technique will sometimes produce bad results. For example, if the reduction goal of a panic mode recovery is a procedure header, the nesting level should be left one greater than the nesting level recorded for the previous state in the parse stack.

There are no obvious methods for handling those cases where the automatic technique for adjusting the status of semantic analysis proves inadequate. Unless such methods are found, the compiler writer will be forced to provide special codes to handle those cases. Because of the nature of the problem, those codes will be dependent on data structures created by the parser and the error handling routines. Since one of the major advantages of using an automatic parser generator is that the user need not understand the data structures it produces, that solution is unsatisfactory.

One final note: panic mode should never be invoked as a result of a semantic error. Panic mode is a last resort method for getting the parser back on track after a syntax error. Since semantic errors are detected as a result of reductions, the parser must be in a legal configuration for a semantic error to be detected. Therefore, there is no reason to invoke panic mode for semantic errors.

8
An Implementation and Empirical Results

The Pascal auditor has been used as a testbed for the error recovery techniques described in previous chapters. Many of the techniques described earlier have been implemented and evaluated as a part of the Pascal auditor. The Pascal auditor also provides empirical evidence of the power and the practicality of the error recovery techniques presented herein. The recoveries produced by the final version of the Pascal auditor have been compared with those produced by Berkeley Pascal [GHJ79], the Burke-Fisher system [BF82], and a version of the Pascal auditor that ignores semantic information during error recovery.

8.1 The Bison Parser Generator

The parser for the Pascal auditor was produced using a new parser generator named Bison. Bison was written to provide support for semantics-directed error recovery. Originally, an attempt was made to adapt the Yacc [Joh78] parser generator. Yacc proved hard to modify because the codes for the various functions it performs are closely intertwined. Therefore, that effort was abandoned, and Bison was written.

Bison is an LALR(1) parser generator. It is based on the DeRemer-Pennello algorithm for computing LALR(1) lookahead sets [DP82]. It is more modular than Yacc, making it easier to modify. The major functional differences between Yacc and Bison are

1. Bison directly supports the division between semantic guards and semantic actions discussed in Section 4.3.
2. Bison generates the additional tables needed for the stack restoration scheme discussed in Section 6.4.
3. The parse tables are organized differently to permit faster access. The tables are slightly larger than those produced by Yacc, but the resulting parsers are faster.
4. A Bison parser maintains an additional stack, the *location stack*, to keep track of information needed for reporting errors and semantic data used in recoveries.

Bison is faster than Yacc because it uses more efficient algorithms for generating the states of the LR(0) automata and for computing the lookahead sets.

The general format of a rule in Bison is

$$\begin{array}{l} \textit{lhs}\text{: } \textit{symbol}_1 \ldots \textit{symbol}_n \\ \qquad [\ \%\text{guard } \textit{expression}\ ] \\ \qquad [\ \textit{action}\ ] \end{array}$$

where *lhs* is a nonterminal symbol, $symbol_1, \ldots, symbol_n$, $n \geq 0$, are symbols, *expression* is a C expression, and *action* is a C compound statement. The square brackets are not part of the rule; they indicate that the enclosed text is optional. The rule causes the production $lhs \rightarrow symbol_1 \ldots symbol_n$ to be part of the grammar accepted by Bison. The semantic attributes of the symbols can be referenced in the guard and action clauses using the $-conventions of Yacc; *i.e.*, the attribute of *lhs* is denoted by "$$" and the attribute of $symbol_k$ is denoted by "$*k*". Information in the location stack about $symbol_k$ is denoted by "@*k*".

The guard expression for a rule checks for semantic errors. The parser initially sets the global variables *yyerror* and *yycost* to zero. If an error is detected, the guard must set *yycost* to a positive value. The compiler writer selects the value based on his estimation of the severity of the error (more severe errors should be assigned higher costs). If the error is one that the compiler writer thinks might be fixed by a syntactic repair, he should have the guard set *yyerror* to one. The guard expression should have no side effects other than setting the values of *yyerror* and *yycost*.

The value of a guard expression is always ignored. Therefore, guard expressions could have been allowed to be statements as well as just expressions. In some cases, restricting guard expressions to be expressions requires a function call to be used where a simple statement would suffice.

The following rule is taken from the grammar for the Pascal auditor.

```
arrval : name
         %guard
            chkarrname($1)
            { $$ = mkarrname($1, &(@1)); }
```

The symbol *arrval* is the name of the nonterminal for array variables. The symbol *name* is a nonterminal symbol that denotes a defined identifier. Another rule reduces an identifier to a *name* and checks if it is defined. If the identifier is defined but is not defined to be an array variable, *chkarrname* will set *yyerror* to 1 and *yycost* to either 5 if the identifier is a subprogram name, or 20 otherwise. The function *mkarrname* constructs a semantic attribute containing the information about the array name that might be needed by later semantic routines. The reference to the location stack entry for the name is used when generating error messages.

8.2 The Parser

A Bison parser automatically performs many of the functions that are needed to support error recovery. An error message should report the location at which the error appears to have occurred. Therefore, it is necessary to keep track of the locations within the input text corresponding to the states in the parse stack. Bison parsers automatically maintain this information in the location stack. After a recovery, the next symbols to be read will be stored in a buffer. Whenever a Bison parser needs another symbol, it first checks if the buffer is empty. If the buffer is not empty, the next symbol will be taken from the buffer; otherwise, the lexical analyzer will be called. Bison parsers perform LR(2) error checking as described in Section 6.4. Because Bison parsers support these functions, the compiler writer is freed from having to supply codes for them.

A Bison parser maintains three separate stacks: a parse stack, a semantic stack, and a location stack. The parse stack contains the states that represent the left context of the parser's current state. The semantic stack contains the semantic values of the symbols that the parser has shifted. The location stack contains information used by the error recovery system.

The location stack maintains information about the locations of the symbols represented in the parse stack. It also contains the timestamp and nesting level information needed to reverse the effects of symbol table operations. Each location stack entry is a record of the form

```
record
    timestamp:      integer;
    nesting_level:  integer;
    first_line:     integer;
    first_column:   integer;
    last_line:      integer;
    last_column:    integer;
    text:           string;
end
```

where *string* is the name of a type used to represent arbitrary length strings. The *timestamp* and *nesting_level* fields are set to the current values of the global timeclock and nesting level counter when the parser shifts over the symbol corresponding to the location table entry. For terminal symbols, the remaining fields must be set by the lexical analyzer. For nonterminal symbols, the parser will automatically set the remaining fields.

The lexical analyzer must set the location and text fields for each token. The *first_line* field must be the number of the line in which the token begins, the *first_column* field must be the number of the column in which the first character of the token appears, the *last_line* field must be the number of the line in which the token ends, and the *last_column* field must be the number of the column in which the last character of the token appears. The *text* field must be a copy of the string representing the token in the input text. The string stored in the *text* field is used as the token's name in error messages.

The parser sets the location and text fields for each entry corresponding to a nonterminal symbol at the time it does the reduction that produces that symbol. Normally, the *first_line* and *first_column* fields are set to the values of the corresponding fields of the first symbol in the handle of the reduction, and the *last_line* and *last_column* fields are set to the values of the corresponding fields of the last symbol in the handle. Reductions according to λ-rules constitute a special case. For a symbol produced by a λ-rule, the *first_line* and *first_column* fields are set to the values of the corresponding fields of the lookahead token, and the *last_line* and *last_column* fields are set to the values of the corresponding fields of the entry at the top of the location stack. The *text* field of a location stack entry that is associated with a nonterminal symbol is set to the null string.

The table organization by Bison parsers is based on the table packing technique proposed by Ziegler and described by Tarjan and Yao in [TY78]. The principal advantage of the table format is that the resulting parsers are fast. Parsing speed is important for error recovery because a piece of text may have to be reparsed several times to test potential repairs. However, the parse tables are also small. The parse tables for the Pascal auditor are 4,764 bytes long. The tables used for stack restoration are another 3,326 bytes, for a total table size of 8,090 bytes. By comparison, the parse tables for Berkeley Pascal are 12,816 bytes long. Much of that space is wasted, since four bytes are used to hold integer values that would fit in two byte integers. If two byte integers had been used, the tables for Berkeley Pascal would occupy 6,408 bytes. If the stack restoration scheme used by the Pascal auditor were replaced by LR(1) pretesting, the tables for the Pascal auditor would be smaller than those for Berkeley Pascal even if Berkeley Pascal's tables had been encoded as two byte integers.

8.3 The Pascal Auditor's Error Recovery System

This section and the two following sections describe the error recovery system used in the final version of the Pascal auditor. They tell how the various components presented in earlier chapters are organized within the system. The local recovery algorithm is described in detail so that the reader may judge which of the improvements claimed for the system are due to the new techniques used in the error recovery system, and which are artifacts of the implementation.

The Pascal auditor invokes its error recovery system whenever the parser detects a syntax error and whenever a semantic guard sets the global variable *yyerror* to one during normal compilation. If the error is a syntax error, the parser backs up the configuration of the parse stack before invoking the error recovery system. The token that caused the error to be detected will then be the second symbol in the lookahead buffer. If the error is a semantic error, no backup is done.

The error recovery system begins by testing if the lookahead buffer is full. If it is not, the algorithm will repeatedly call the lexical analyzer to fill the buffer. The system then checks if the error was a syntax error or a semantic error. The variable *yyerror* will be zero if the error was detected by the parser, and one if it was detected by a semantic guard. A global flag, named *semantic_error*, is set to one for semantic errors and zero for syntax errors. The flag is needed because *yyerror* will be reset by semantic guards while testing potential repairs. The local recovery algorithm is then invoked. If the local recovery algorithm produces a repair, the error recovery system returns and normal parsing resumes. If no repair is found and the error was a semantic error, the error recovery system performs the semantic action associated with the rule that caused the error to be detected; otherwise, panic mode is invoked.

The local recovery algorithm used by the Pascal auditor is based on the Graham-Haley-Joy algorithm [GHJ79]. There are two major differences between the Pascal auditor's local recovery algorithm and the Graham-Haley-Joy algorithm. The algorithm used by the Pascal auditor uses general static semantic data to direct the choice of a repair. The Graham-Haley-Joy algorithm uses some semantic data, but that data is represented syntactically (see Chapter 3). Also, the potential repairs considered by the two algorithms are different.

The local recovery algorithm starts by doing a forward move over the text in the lookahead buffer. If the error was detected semantically, the initial forward move establishes the cost of making no change to the input text. The semantic cost of any potential repair is the sum of the values assigned to *yycost*. The local recovery algorithm will not consider a repair whose semantic cost equals or exceeds that initial cost. The algorithm then computes the LR(1) lookahead set for the top state of the parse stack using the function *Shiftable* (see Section 6.3). If the first symbol in the error lookahead buffer is a member of that lookahead set, the LR(1) lookahead set for the configuration reached after shifting that token is also computed. The lookahead sets are used to quickly eliminate some infeasible repairs from consideration. The local repair algorithm then applies its repertoire of potential repairs. Syntactic and semantic costs are computed for each repair found to be feasible. If a repair costs less than any previous repair that was found to be feasible, it is logged as the current repair of choice. Finally, if any repairs were found to be feasible, the least costly of those repairs is applied.

Testing a potential repair is a two step process. Repairs are implemented by modifying the contents of the lookahead buffer. A potential repair is tested by making a copy of the lookahead buffer as it would appear after applying the repair, and then parsing over that copy. That parse is the forward move. However, performing semantic analysis while doing a forward move is time consuming. Therefore, a preliminary forward move that consists of parsing without semantic analysis is done. If the preliminary forward move determines that the syntactic cost of the repair is too great to allow it to be chosen as the repair to be applied, the forward move with semantics is not done. This use of parsing to eliminate infeasible repairs is the opposite of the situation in the Berkeley Pascal, which uses semantic information to reduce the time spent parsing. Since the semantic information that is used by the Berkeley Pascal is encoded in the nonterminal symbols, there is little cost associated with the semantic checks. The Pascal auditor's use of semantic routines is more time consuming. However, because of its superior encoding of the parse tables, the bare bones parsing algorithm used for the preliminary parse runs from 5 to 8 times faster than the parser used by Berkeley Pascal. Empirical results show that the preliminary parse significantly improves the speed of the local recovery algorithm.

The cost of a repair has three components. The dominant component is the distance parsed before a new syntax error is detected. If fewer than three tokens can be shifted, the repair is automatically rejected. The repair will also be rejected if the forward move does not shift at least as far as the best repair found so far. The other components of the cost are the semantic cost and the syntactic cost. The semantic cost of a repair is the sum of the values of *yycost* after every evaluation of a semantic guard done during the forward move. The Pascal auditor favors repairs whose semantic cost is zero over those with positive semantic costs. Recall that a positive semantic cost means that at least one semantic error has been found during the forward move. Since most errors are either purely syntactic or purely semantic, a syntactic repair whose semantic cost is positive is probably undesirable. If a repair has a positive semantic cost and it is not otherwise eliminated from consideration, its total cost for purposes of comparison with other repairs is formed by adding its syntactic and semantic costs.

The syntactic cost of a repair is computed using cost functions. There are three cost functions: *Icost*, *Dcost*, and *Rcost*. *Icost* takes a symbol as its argument and returns the cost of inserting that symbol. *Dcost* computes the cost of deleting a symbol. It takes two symbols as arguments. The first symbol is the symbol to be deleted, and the second is the symbol that precedes it. If the two arguments represent the same symbol and the symbol is one without associated semantics, the cost returned is one. This special case is included as a heuristic. The function *Rcost* returns the cost of replacing one symbol by another. *Rcost* takes three arguments. The second argument is the symbol to be replaced, and the first is the symbol replacing it. The third argument is used only when the symbol to be replaced is an identifier. It is the character string that forms the identifier. A spelling matching algorithm is used to decide if the string is close to the spelling of a keyword. If it is, the cost of the replacement is reduced to one.

There is a special cost called ***infinity***. If the syntactic cost of a repair is infinite, that repair is never attempted. The only deletion for which infinity is returned is deletion of an end-of-file symbol. Some insertions can never be chosen as the final repair. For example, in Pascal, the symbol '=' is permitted in every context in which any relational operator other than '**in**' is permitted. It is natural to prefer '=' over the other relational operators when testing insertions. Hence, there is no reason even to try inserting the other five relational operators. Therefore, to save time, the insertion costs of the other relational operators are infinity. The infinite cost has its greatest value for replacements. Most replacements should not be allowed under any circumstances. For example, it is unlikely anyone would ever accidently type the keyword **procedure** where he meant to type the symbol ':='. The current version of the Pascal auditor attempts too many unreasonable replacements, which wastes time and occasionally results in an unreasonable recovery.

Some repairs are combinations of simpler repairs. For example, the Pascal auditor sometimes tries deleting two consecutive tokens. In such cases, the syntactic cost of the repair is the sum of the costs of the simpler repairs.

Certain common errors are handled by error productions. The grammar for the Pascal auditor has been extended to allow declarations to appear in any order. Other error productions permit general type specifiers to appear wherever a type identifier or ordinal type can appear. Also, the grammar has been extended to allow an expression to appear in most contexts where a constant can appear. An exception was made in the case of the bounds of subranges because the error production for that case caused an LR(1) conflict. The semantic routines for error productions produce the error messages for the errors handled by those productions.

8.4 The Repairs

The Pascal auditor considers four types of repairs: deletions, insertions, replacements, and bracket repairs. Deletion consists of removing one or two tokens from the error lookahead buffer. The Pascal auditor first tries deleting the second symbol in the lookahead buffer and then tries deleting the first symbol in the buffer. If the error was detected semantically no further deletions are attempted. Otherwise, the Pascal auditor tries deleting the second and third tokens, and finally, it tries deleting the first and second tokens. The deletions are tested in this order because the Pascal auditor does LR(2) error checking. With LR(2) error checking, it is more likely that the second token in the buffer should be deleted than the first. The two token deletions are not tried for semantic errors because they are too likely to lead to inaccurate repairs. Consider the statement

$$x := 1$$

where x has been declared to be a parameterless procedure. The semantic guard for the destination of an assignment causes the error recovery algorithm to be invoked. If the syntactic repair algorithm tries deleting the symbols ':=' and '1' simultaneously, it will find that the resulting program is both syntactically and semantically correct. It will, therefore, choose to apply that repair unless the costs of the deletions are set prohibitively high. However, in this case, it is better to leave the original text unchanged and report the semantic error.

Double deletions have proven to be important repairs for Pascal. It is a common error to include an empty pair of parentheses in subprogram headers and calls. Without the ability to delete both parentheses, the recovery algorithm would be forced to resort to panic mode. If the error is in a procedure or function header, applying panic mode could lead to many spurious errors later.

The insertions considered by the Pascal auditor consist of inserting single tokens into the lookahead buffer. The tokens that it will try inserting are the elements of the LR(1) lookahead sets computed at the start of the syntactic repair algorithm (see the previous section). It first tries inserting tokens after the first token in the lookahead buffer. Next, it tries inserting tokens at the start of the lookahead buffer.

The replacements consist of substituting a token for one of the tokens in the lookahead buffer. Replacements for the first and second symbols in the buffer are considered. The LR(1) lookahead sets are used to restrict the replacements that are attempted to those that might be feasible.

Bracket repair is a new type of repair. Consider the statement

$$a(i) := k;$$

where a is an array of integers, and i and j are integer variables. The likely error is that parentheses have been used in place of square brackets. The Burke-Fisher system responds to this error by replacing ':=' with ';', *i.e.*, it converts the assignment statement into a pair of procedure statements. This repair is the natural choice for the Burke-Fisher algorithm because it makes no use of semantic data when selecting syntactic repairs and therefore is unable to take advantage of the fact that neither a nor j is the name of a procedure. However, Berkeley Pascal, which does know that a is an array variable and j is a variable, does little better. It reports that a procedure name was expected where a appears and that the statement is malformed. The reason Berkeley Pascal reports that it expects a procedure name is that, in Berkeley Pascal, it costs less to replace an array name with a procedure name than it does to replace a left parenthesis with a left square bracket. Moreover, even if the costs were revised, the results for this example would remain the same. When Berkeley Pascal tries replacing the left parenthesis with a left square bracket, it rejects the repair because of the error it detects upon reaching the right parenthesis.

The best repair for a bracketing error often involves a pair of insertions or replacements. In the previous example, the left parenthesis must be replaced by a left square bracket, and the right parenthesis by a right square bracket. The bracket repairs done by the Pascal auditor consist of inserting a left bracket or replacing a symbol by a left bracket near the detection point of an error. Then, if a new syntactic error is detected during the forward move, the corresponding right bracket is used in insertions and replacements near the point of the new error.

The left parenthesis ('('), the right parenthesis (')'), the left square bracket ('['), and the right square bracket (']') are the only symbols treated as brackets by the Pascal auditor. Other pairs, such as **begin** - **end** and **repeat** - **until**, are also brackets in the usual grammatical sense. However, users are far less likely to make mistakes with those brackets. For example, there is little chance of accidentally substituting **repeat** - **until** for **begin** - **end**.

The algorithm for bracket repairs is more complex than those for the other repairs. In fact, the code for implementing bracket repairs is longer then the codes for the other repairs combined. The algorithm works as follows. For each left bracket, four repairs are considered. The bracket can be inserted before the first token in the lookahead buffer, it can be inserted after that token, it can replace the first token in the buffer, or it can replace the second token. For each of those repairs, the follow sets are consulted to decide if the repair might be viable. If a repair is viable, a copy of the modified buffer is parsed to check for subsequent syntax errors. If no such error is found, the bracket repair algorithm does not give any further consideration to the repair, since it will already have been considered as a possible insertion or replacement. If a new syntax error is found, the copy of the lookahead buffer is further modified in an attempt to repair the second error. Attempts will be made to insert the corresponding right bracket immediately before the detection point of the second error, and also before the symbol preceding the detection point. Attempts will also be made to replace the symbols on either side of the detection point with the right bracket. Thus, as many as sixteen repairs will be considered for each bracket pair. As a heuristic, if a left bracket was inserted, the cost of inserting the right bracket is reduced, and if the left bracket replaced another token, the cost of replacing another token with the right bracket is reduced.

The algorithm described does not always find the best repair for a bracketing error. Indeed, it does not always find a repair at all. Sometimes, the second error will not be detected until after the point where it actually occurred. Consider the statement

$$a[i] := a(i + a[j]$$

where a is an array of reals, and i and j are integers. The error here is two-fold. First, the left parenthesis should have been a left square bracket. Second, there should be a right square bracket between the second instance of i and the '+'. However, when the bracket repair algorithm tries replacing the left parenthesis with a left square bracket, it does not find a subsequent syntax error until it reaches the semicolon. Therefore, the second repair will consist of inserting a right square bracket immediately before the semicolon. This repair will, of course, lead to a semantic error since real expressions cannot be used as subscripts. An earlier version of the bracket repair algorithm avoided this problem. That version did not limit its attempts to fix the second error to changes made in the immediate context of the detection point of the second error. Instead, it tried replacements and insertions in every position between the first repair and the second error's detection point. Errors such as the one shown above posed no problem for that version of the bracket repair algorithm. However, it was dropped when it was discovered that the error recovery algorithm was spending about two-thirds of its time testing bracket repairs. The current algorithm spends only about one-tenth of its time testing bracket repairs and yet produces the same results for all bracketing errors that occur in the Ripley-Druseikis sample.

8.5 Reporting Errors

The output of the Pascal auditor is a listing of the input program with interspersed error messages. The nature of an error message depends on how the message was generated. Error messages produced by the local recovery algorithm indicate the changes made to the input text by the algorithm. The compiler writer must make provisions for producing error messages for semantic errors, error productions, and panic mode recoveries. Information matching locations in the input text with symbols involved in error messages can be obtained from the location stack. For panic mode recoveries, the location stack entry for the **error** token is set so that its starting location is the start of the text skipped over during the recovery, and its ending location is the end of that text.

The error messages produced by the Pascal auditor may be associated with either a single point in the program text or a contiguous region of the program. A error message associated with a single point in the program text is indicated by a caret pointing to that location. An error message associated with a region of the program is marked by angled brackets indicating the endpoints of the region with the space between the brackets filled by hyphens. If the region associated with an error message overlaps the location associated with another error message, the location markers associated with the inner error are given precedence. For example, suppose a Pascal program contains the statement

$$x := [y + 2]$$

where x and y are real variables. Then the listing the Pascal auditor produces will include the following error report:

```
                 x := [y + 2]
                 <-----<--->>
***     3:  e - incompatible assignment
***     9:  e - set member type is not ordinal
```

Every error message generated by the Pascal auditor is written to a temporary file called the *error file*. After analysis of a program has been completed, the error file is sorted so that the error messages appear in the order they are to appear in the listing. The error file and the input file are then rewound, and the listing is produced from them.

8.6 Space and Time

The Pascal auditor was written in C [KR78] on a Digital Equipment Corporation VAX-11/780† running Berkeley UNIX‡. It accepts full ANSI Pascal [ANS83]. The grammar used in the implementation of the Pascal auditor is shown in Appendix A. The source code for the Pascal auditor is 18,567 lines long. The sizes of its major components are as follows:

lexical analyzer	1205 lines	6%
grammar	1127 lines	6%
error handler	3697 lines	20%
semantic routines	11572 lines	62%

The remaining 6% of the source code consists of header files, the main routine, and utility routines. The compiled code (including tables) is about 120,000 bytes long.

For error-free programs, the Pascal auditor is about as fast as Berkeley Pascal. The Pascal auditor analyzes a correct program in about two-thirds the time it takes the

† VAX is a registered trademark of Digital Equipment Corporation.

‡ UNIX is a registered trademark of Bell Laboratories.

Berkeley Pascal interpreter to analyze and produce interpretive code for the same program. Using the profiler gprof [GKM83], it was ascertained that, for error-free programs, the Berkeley Pascal interpreter spends about two-thirds of its time analyzing the programs; the rest of the time is spent in code generation. The profiler further showed that relative times the two systems spend in the various phases of analysis are quite different. Berkeley Pascal spends a higher percentage of its time parsing and performing lexical analysis. The Pascal auditor spends most of its time performing semantic analysis. One reason the Pascal auditor takes longer to perform semantic analysis is that the semantic routines must obey the restrictions described in Sections 5.3 and 5.4. It is sometimes necessary to use less efficient codes for semantic analysis than could be used in the absence of those restrictions. However, the major reason semantic analysis takes longer appears to be the division of semantic routines into guards and actions that was discussed in Section 4.3. If the alternative scheme for implementing semantic routines that was described at the end of that section had been used, the time required to analyze correct programs would have been significantly less for the Pascal auditor than for Berkeley Pascal.

It is hard to compare the speeds of error recovery systems. For a given program, the systems might choose different repairs for an error that occurs early in the program, which may affect further analysis of the program. To compare the speeds of the Pascal auditor and Berkeley Pascal, a special set of erroneous programs was developed. Those programs had the property that the two systems repaired each error in exactly the same way. In addition, a corrected version of each program was written to provide a control against which the extra time spent performing error recovery could be measured.

Timings show that the Pascal auditor's error recovery algorithm is significantly slower than Berkeley Pascal's. The timings were done using the UNIX time command. The time command produces two times for a program: the user time and the system time. User time is the time spent in the user process, while system time is the time spent performing system commands. The timings for the Pascal auditor show that it spends about 0.09 seconds of user time for each error detected. Berkeley Pascal, on the other hand, spends only about 0.03 seconds of user time for each error. The system time spent by two algorithms is approximately equal; both spend about 0.02 seconds of system time for each error. Thus, the Pascal auditor takes more than twice as long to recover from an error as Berkeley Pascal.

The Pascal auditor is slower than Berkeley Pascal for many reasons. One reason is that the Pascal auditor's error recovery codes are not as efficient as they could be. Efficiency was not a primary consideration in the design of the Pascal auditor. (Neither was it for Berkeley Pascal.) The original version of the Pascal auditor required about six times longer to recover from an error than does the current version. Many improvements have been made to that original version; however, other changes that might have further improved its efficiency were not implemented because they involved major revisions. Another reason the Pascal auditor is slower than Berkeley Pascal is that it considers more repairs for each error. The use of semantic information also contributes to the time spent in error recovery. For each repair found to be syntactically feasible, the repair algorithm performs a forward move that includes semantic evaluation. The time spent creating the appropriate environment for that forward move, doing the semantic actions, and then restoring the previous semantic environment takes much longer than a purely syntactic forward move.

The timings produced one surprise. The correlation between the speed of the error recovery algorithm and the bound on the number of tokens considered during a forward move was found to be less than expected. In the original version of the Pascal auditor, the bound on forward moves was 15 tokens. Reducing that bound to seven tokens reduced the time spent for each recovery by only about 10%. Not surprisingly, when the bound was reduced to seven, the quality of the recoveries produced by the error recovery algorithm suffered. However, the quality of the recoveries produced when the bound was 15 tokens was not as good as when the bound was set in the range from 9 to 12 tokens. When the bound was 15 tokens, there were instances where errors that occurred later in the program adversely affected the costs assigned to the repairs being tested. In the current version of the Pascal auditor, the bound on the number of tokens used in a forward move is 12 tokens.

The timings described above are biased in favor of Berkeley Pascal. Earlier timings showed that speed of Berkeley Pascal was less than double that of the Pascal auditor. Unlike the sample used in the timings described above, the sample used in those earlier timings contained programs for which different repairs were produced by the two algorithms. The Pascal auditor found viable repairs in many cases where Berkeley Pascal did not. When Berkeley Pascal or the Pascal auditor fails to find a repair for an error, it takes longer to recovery from the error because more potential repairs are tested. Also, the error recovery system must spend time executing its panic mode algorithm. Thus, the fact that the Pascal auditor usually tests more potential repairs than Berkeley Pascal is partially compensated for by the fact that it is more likely to find a repair.

```
     1       var a, b: array [1..5 1..10] of integer;
            ^                     ^

***    2:  e - Missing Program Header
***   24:  e - inserted ','
     2           i, j, k, l: integer;
     3     begin
     4       3: i + j > k + l * 4 then go 1 else k is 2;
             ^ ^                       ^  ^        ^

***    3:  e - label 3 is undeclared
***    5:  e - inserted 'if'
***   29:  e - replaced 'go' with 'goto'
***   32:  e - label 1 is undeclared
***   41:  e - replaced 'is' with ':='
     5       a 1,2 := b [ 3 * ( i+4,j*/k]
              ^   ^                ^  ^  ^

***    4:  e - inserted '['
***    8:  e - inserted ']'
***   25:  e - inserted ')' before ','
***   28:  e - deleted '/'
***   31:  e - inserted ';'
     6       if i=1 then then goto 3;
                         ^

***   15:  e - deleted 'then'
     7     2: end.
           ^

***    1:  e - label 2 is undeclared
```

Figure 8.1 The Graham-Rhodes example

8.7 Examples of Use

This section presents two examples of recoveries performed by the Pascal auditor. Other examples may be found in Appendices B, C, and D. The examples presented here were chosen because they are well-known examples from the literature.

The example shown in Figure 8.1 is taken from the paper by Graham and Rhodes [GR75]. Variants of this example appear in [PD78], [GHJ79], and [BF82]. The text shown in Figure 8.1 is taken from a listing produced by the Pascal auditor. Each line of the source text is preceded by its line number. The lines prefixed by three asterisks are error messages. The number following the asterisks is the column number of the start of the region associated with the error message.

The error messages produced for the Graham-Rhodes example are as good as those that an expert programmer checking the program for errors might be expected to produce. However, semantics affected the choice of repairs in only one case. The recovery algorithm would not have chosen to delete the slash ('/') that appears in column 28 of line 5 had semantic information been ignored. The cost of deleting the slash is greater than the cost of inserting an identifier between the star ('*') and the slash. Hence, if semantic information were ignored, the local recovery algorithm would have chosen to repair the error by inserting an identifier. That repair was not chosen because the expression in which the slash occurs is a subscript expression. The type of the subscript is known to be integer; however, in Pascal, the result type of the operator '/' is real. Therefore, simply inserting an identifier leads to a semantic error.

```
        1     program sillypascal(input, output);
        2     var
        3       mychar: char;
        4     begin
        5       read mychar;
                    ^       ^
***     7:    e - inserted '('
***    14:    e - inserted ')' before ';'
        6     end.
```

Figure 8.2 P. J. Brown's example

Figure 8.2 shows the result of applying the Pascal auditor to P. J. Brown's example [Bro82, Bro83]. When this example is presented to the version of the Pascal auditor that ignores semantic information during error recovery, it recovers from the syntactic error by inserting a semicolon (';') between the procedure *read* and the variable *mychar*. As a result of that repair, two semantic errors are also reported. The message "missing parameter list" is given following the function *read*, and the variable *mychar* is flagged with the message "a variable appears where a procedure was expected."

8.8 Comparisons

The goal of this work has been to develop practical error recovery techniques that diagnose errors more accurately than do earlier techniques. To measure the success of this work, the recoveries produced by the Pascal auditor have been compared with those produced by Berkeley Pascal [GHJ79] and the Burke-Fisher system [BF82]. Berkeley Pascal's error recovery system is perhaps the best system yet to be included in a production compiler. The Burke-Fisher error recovery system was developed as an enhancement of Berkeley Pascal's system. The Burke-Fisher system has been used in some experimental compilers.

The major innovation of this work is the use of general static semantics to aid in error recovery. Testing the Pascal auditor against earlier error recovery systems provides evidence of the advantages of semantics-directed error recovery, but that evidence is muddied by other differences among the various systems. To provide clearer evidence of the benefits of using semantics-directed error recovery, a version of the Pascal auditor in which semantic information is ignored during error recovery has been created. The recoveries produced by that version of the Pascal auditor have been compared with the recoveries produced when semantic information is used.

Some of the listings used in the comparisons are reproduced in Appendices B, C, and D. Appendix B contains the listings for every program in the test sample where the recoveries chosen by the version of the Pascal auditor that ignores semantic data differ from the recoveries chosen by the version of the Pascal auditor which uses semantics. Appendix C contains the listings for every program where the recoveries produced by Berkeley Pascal or the Burke-Fisher system are better than those produced by the Pascal auditor. Appendix D contains the listings for some of the programs for which the Pascal auditor outperforms Berkeley Pascal.

The test sample used in the comparisons is a modified version of the Ripley-Druseikis sample [RD78]. The Ripley-Druseikis sample consists of 126 Pascal programs that demonstrate a variety of errors. Unfortunately, the programs contained in the original sample are incomplete. In particular, most declarations are missing. The programs in the modified sample include all necessary declarations.

The comparison with Berkeley Pascal produced impressive results. The Pascal auditor produced better recoveries than did Berkeley Pascal for 43 of the programs in the test sample. The Pascal auditor's recoveries were inferior to those of Berkeley Pascal for only seven programs.

Six of the cases where Berkeley Pascal produced better recoveries than the Pascal auditor did not involve semantics. Berkeley Pascal is sometimes able recover from an error by inserting two tokens. That capability accounts for two of the instances where Berkeley Pascal outperforms the Pascal auditor. Berkeley Pascal treats some multiple character symbols as sequences of tokens. In particular, the symbol ':=' is treated as the token ':' followed by the token '='. That feature accounts for three of those instances where Berkeley Pascal bests the Pascal auditor. Berkeley Pascal's lexical analyzer treats the symbol '?' as a special quote symbol. An error message is given for a string delimited by '?', but the lexical analyzer recognizes it to be a string. Treating '?' as a string quote accounts for one instance where Berkeley Pascal bests the Pascal auditor.

The Pascal auditor could be modified to produce the same recoveries as Berkeley Pascal in the cases mentioned above. Berkeley Pascal's mechanism for inserting multiple tokens could be copied in the Pascal auditor. The Pascal auditor could also be modified to treat the symbol ':=' as two separate tokens and to treat the character '?' as a string quote. However, those changes probably are not desirable. Treating the symbol ':=' as two separate symbols causes bad recoveries for the examples in the Ripley-Druseikis sample as often as it allows good recoveries that could not have been performed otherwise. Furthermore, there are some errors for which treating ':=' as two symbols allows Berkeley Pascal to perform apparently strange recoveries. For example, consider the following listing produced by Berkeley Pascal:

```
         1  program p;
         2  var i: integer;
         3  begin
         4    8 := 0
e -----------^--- Replaced '=' with a keyword goto
         5  end.
E 4 - 8 is undefined
E 4 - 0 is undefined
In program p:
  w - variable i is neither used nor set
```

The likely error in this example is that the programmer wrote the digit 8 where he meant to put the variable i. The recovery produced by Berkeley Pascal seems more likely to confuse than inform most programmers. Treating the character '?' as a string quote without regard to the context in which it appears seems more likely to lead to poor recoveries than to good ones.

There was one case where using semantics to assist in error recovery led to an inferior recovery. The program in that case was as follows:

```
program p(input, output);
var prcount, x: integer;
begin
99 prcount := prcount;
   x := 1
end.
```

It seems likely that the programmer intended that the integer 99 should be a label. However, because 99 was not declared to be a label, the Pascal auditor detects a semantic error when it tries inserting a colon between 99 and *prcount*. Therefore, it rejects that repair in favor of deleting the integer 99. Berkeley Pascal, the Burke-Fisher system, and the Pascal auditor with semantics disabled all patched this error by inserting a colon between 99 and *prcount*. Since the programmer probably intended to place a colon at that location, the recovery chosen by those systems is better than the one chosen by the Pascal auditor. There are no other programs in the Ripley-Druseikis sample where using semantic data causes an inferior recovery to be selected. A better example, which does not appear in the Ripley-Druseikis sample, would be if a goto-statement referring to the label 99 appeared following the text examined during the forward move. The same recovery would be produced for that example.

Those instances where the Pascal auditor outperformed Berkeley Pascal were also analyzed. No one factor accounts for more than seven of those instances. The major causes for the improvement include

1. The use of general static semantic information. Berkeley Pascal uses some semantic data during error recovery, but it does not take advantage of all of the static semantic information that is available.
2. The difference in the weights assigned to semantic information. The cost of replacing one type of identifier with another in Berkeley

Pascal appears to be too low.

3. The use of LR(2) error checking. Berkeley Pascal is sometimes unable to find the best recoveries because it has performed some erroneous reductions.
4. The Pascal auditor's bracket repair capability.
5. The use of the spelling matcher for keywords.
6. A better panic mode algorithm. This result is surprising since the panic mode algorithm used by Berkeley Pascal was hand tailored.

The Pascal auditor outperformed the Burke-Fisher system for 31 programs of the test sample. However, the reason the Pascal auditor did that well was that Burke and Fisher did not use any error productions. Discounting those examples where the improvement was due to error productions, the Pascal auditor did better than the Burke-Fisher algorithm for 24 programs. There were only six programs for which the Burke-Fisher system produced better recoveries than did the Pascal auditor.

The six programs for which the Burke-Fisher system produces better recoveries than does the Pascal auditor have been analyzed to determine the reasons for differing recoveries. Like Berkeley Pascal, the Burke-Fisher system is able to insert two tokens in special circumstances. That capability accounts for two of the cases where the Burke-Fisher system produces a better recovery than the Pascal auditor. The Burke-Fisher considers a repair to be viable after a very short forward move. In one program where there are two unrelated errors in close proximity, the Pascal auditor rejects all repairs because it detects a new error too close to the point of the repair, while the Burke-Fisher system finds a repair because of its shorter parse check. The Burke-Fisher system is able to delete terminal symbols that have been shifted onto the parse stack. That capability leads to a better repair for one program. The Burke-Fisher system is sometimes able to merge two adjacent tokens into a single token. In particular, for one of the programs in the test sample, it is able to merge the identifier *go* and the keyword **to** to produce the keyword **goto**. Finally, as was noted above, the Burke-Fisher system outperforms the Pascal auditor in one case where the use of semantic data leads to an inferior recovery.

Most instances where the Pascal auditor outperforms the Burke-Fisher system stem from the use of semantic information. The Burke-Fisher system considers more types of repairs than either Berkeley Pascal or the Pascal auditor. However, the Burke-Fisher system uses less information to decide which of the potential repairs to apply. Therefore, it often chooses inferior repairs. Consider, for example, the statement

if *nonprime* = 0 **then** *numprime,x.* := *numprime*(*x*) + 1;

where *numprime* is an array of integers and *x* is an integer variable. The Burke-Fisher algorithm repairs the syntactic errors in this example by replacing the comma with a semicolon and inserting an identifier after the period.

The final comparisons were between the Pascal auditor and the version of the Pascal auditor that ignores semantic information during error recovery. These comparisons are the best test of semantics-directed error recovery, since the results are not contaminated by other differences between the two systems. The listings produced by the two versions of the Pascal auditor for every program for which the recoveries differed are shown in Appendix B. There were 27 programs for which different recoveries were produced. The recoveries produced using semantics were better for 21 of those programs. The recoveries produced while ignoring semantics were better in only one case. In the remaining 5 cases, the recoveries were different, but there was little difference in their quality.

9
Implementation Notes

The preceding chapters dealt with the large issues involved in creating an error recovery system. However, much of the time spent writing the Pascal auditor was expended solving little problems. Many of those problems were of such a nature that they must have arisen and been solved for other error recovery systems. However, because the authors of those systems did not report their solutions, new solutions had to be developed from scratch. Solutions to some of the problems encountered while implementing the Pascal auditor are presented here as a guide to others.

9.1 Error Messages for Insertions

The Pascal auditor indicates the location of an insertion more clearly than either Berkeley Pascal or the Burke-Fisher system. All three systems use a single caret to mark the location of an insertion. Berkeley Pascal places the caret so that it points to the first character of the first token following the insertion. The Burke-Fisher system has the caret point to the last character of the last token preceding the insertion. For single character tokens, both schemes might cause confusion. For example, consider the following fragment of a listing produced by Berkeley Pascal

```
          7  begin
          8    p
          9    q
      e --------^--- Inserted ';'
         10  end.
```

In this example, *p* and *q* are parameterless procedures. The likely error is that the semicolon that must follow *p* has been omitted. However, a naive user might believe that the error message meant that a semicolon was inserted after *q*. The Pascal auditor places the caret for an insertion so that it points to white space. The Pascal auditor's output for the previous example is

```
          7      begin
          8        p
                    ^
    ***      4:  e - inserted ';'
          9        q
         10      end.
```

As this example shows, having the caret point to white space leaves no doubt where the insertion occurred. There are cases where there is no white space around the point of an insertion. To avoid ambiguities in those cases, the Pascal auditor's error message indicates which token the insertion preceded, as is illustrated by the following example:

```
        2     var a: array [1+10] of integer;
                                ^
     ***    16: e - inserted '..' before '+'
```

If the phrase "before '+'" were not provided, a naive user might think that the dots were inserted after the '+'.

The placement of a caret indicating the location of an insertion depends on the token being inserted. If the token being inserted is a separator or a single character closing bracket, it is inserted at the first location following the token preceding the insertion. Otherwise, it is inserted at the first location preceding the token following the insertion. Thus, the rules regarding the placement of carets cause the locations indicated for the inserted tokens to conform to common coding conventions.

9.2 The Lexical Analyzer

The Pascal auditor's lexical analyzer is very fast. Its speed is one reason why the Pascal auditor is as fast as it is relative to Berkeley Pascal. There are several reasons for its speed. The routine for reading the input text uses the UNIX system call *read* directly, and the buffer size was chosen to match the system buffer size. Thus, the overheads associated with using the UNIX standard I/O library were avoided. To avoid unnecessary copying, tokens are represented by pointers into the input buffer whenever possible. Also, the lexical analyzer was coded in a style that avoids unnecessary procedure calls.

The lexical analyzer's handling of semantic values can be improved. The semantic value of an identifier is the string that represents that identifier. The semantic action routines are responsible for looking up identifiers in the symbol table. Hence, when evaluating the semantics associated with a possible repair, the local recovery algorithm must look up the identifiers encountered during the forward move. The result is that the error recovery routines spend about 12% of their time looking up identifiers. If the semantic values of identifiers were made to be pointers to the associated symbol table entry, that time could be saved.

Unmatched string quotes are among the hardest lexical errors to handle well. In Pascal, a string is not allowed to extend past the end of a line. Therefore, if a line contains an odd number of string quotes, there must be a lexical error. Many Pascal compilers check for unmatched quotes only at the end of a line. If there are unmatched quotes, a string is formed from the text from the last quotation mark to the end of the line (an implicit quote assumed to exist at the end of the line). The text absorbed into the string will often contain the tokens terminating the statement in which the quotation mark appears. Thus, this manner of handling unmatched quotes sometimes interferes with the analysis of the text on the following line.

Some new heuristics for handling unmatched string quotes have been applied in the Pascal auditor. Whenever a quotation mark is encountered, the number of quotation marks to the right of it on the same line are counted. If no other quotation marks are found, the original quotation mark is assumed to be unmatched. If an even number of quotation marks are found on the rest of the line, there must be an error. A check is made to see if the token preceding the original quotation mark can legally precede a string. If not, the quotation mark is assumed to be unmatched. An unmatched quotation mark is returned as an illegal token; no effort is made to construct a string starting from it. This heuristic often allows the Pascal auditor to handle unmatched string quotes more gracefully than if the simpler seheme described above had been used. It is especially helpful is those circumstances where the programmer did not intend to write a string at all, but simply made a typographical error.

The heuristic used by the Pascal auditor could be improved by checking if the token following a supposed string can legally follow a string. For example, consider the statement

writeln(' x = , *x*, ' y = ', *y*)

The apparent error is that a quotation mark is missing between the first equal sign and comma; that is, the first quotation mark is unmatched. The heuristic currently used by the Pascal auditor will cause it to decide that the last quotation mark is unmatched. If the token following the string were checked, then it would be clear that the first quotation is the one that is unmatched since the identifier *y* cannot immediately follow a string. Therefore, the lexical analyzer would return an illegal character token for the first quote. Eventually, the panic mode algorithm would report that the tokens between the open parenthesis and the first comma constitute a malformed expression.

9.3 Assigning Costs to Syntactic Repairs

The costs of syntactic repairs in the Pascal auditor were based on intuition and experimentation. The costs are small positive integers (see Section 8.3). Initially, uniform costs were assigned to each class of repair. The cost of insertions was 3, the cost of replacements was 5, and the cost of deletions was 7. Those costs were chosen because, for a sample of erroneous Pascal programs, insertions were the best repair most often, replacements next most often, and deletions least often. The costs were then refined for the test sample to eliminate undesirable recoveries. Experience gained while refining the costs showed that the best choice of costs for repairs often did not correspond to the relative frequencies that those repairs were optimal.

For most errors, the relative costs of particular insertions and deletions have little impact on the recoveries. Suppose that an error has been detected. Suppose further that inserting or deleting a single token near the error's detection point eliminates all detectable syntax and semantic errors in the surrounding text. Then that repair is almost always as good a recovery as could be expected. Even when there are many possible insertions or deletions that could repair an error so that no further errors are discovered, there is usually little reason to prefer one of those repairs above the others.

The costs assigned to replacements strongly affect the quality of recoveries. The costs of replacements should almost always be greater than the costs of insertions or deletions. In fact, most replacements should be prohibited. For example, there is an erroneous program for which Berkeley Pascal repairs an error in the program by replacing the operator '+' with the keyword **label**. The chosen repair subsequently causes a spurious semantic error to be reported. Worse than just being inaccurate, the repair looks foolish. There is almost no chance that someone would accidentally write '+' where he meant to write **label**. Experience indicates that initially all replacements should be assigned prohibitively high costs. The cost of a particular replacement should not be lowered until an example is found where that replacement is the best repair. In addition to preventing some seemingly foolish recoveries, banning most replacements reduces the number of repairs considered during error recovery, thereby making the error recovery system faster.

The cost assigned to a replacement usually should be greater than the cost of deleting the symbol being replaced and the cost of inserting the symbol replacing it. Berkeley Pascal's convention of setting the cost of the replacement equal to the sum of those other two costs seems a good idea. Errors for which the optimal repair is a replacement are common. Thus, it might seem to be a mistake to assign high costs to replacements. That notion is, however, incorrect. Replacements are usually the repair of choice only if there is no simpler repair that is syntactically and semantically viable. If inserting or deleting a single token repairs an error, then the chances are that that repair is better than any replacement. There are, naturally, some exceptions to this rule. For example, in Pascal, the cost of replacing a semicolon with a colon should be small, but the cost of deleting a semicolon should be large.

Some special cases were found where the costs of some deletions should be lower than the costs of some insertions. In particular, it was found that the cost of deleting a right bracket should be lower than the cost of inserting the corresponding left bracket. Suppose the parser discovers an extra right bracket. The chance that the programmer intended to write another left bracket is probably about equal to the chance that he meant to write fewer right brackets. However, even if the error recovery system could determine that another left bracket should be inserted, it usually would still be unable to determine where to insert it. On the other hand, deleting the unmatched right bracket has a good chance of being the optimal repair. Thus, if an unmatched right bracket is found, the chance that deleting the right bracket corresponds to the programmer's intent is usually greater than the chance that any particular insertion of a left bracket corresponds to his intent.

The costs of replacing an illegal character with legal tokens should depend on the particular character. Suppose an illegal character appears in a Pascal program. The cost of replacing that character by tokens represented by characters that are near it on most keyboards should be less than the cost of replacing it with tokens represented by characters that are far away from it. This idea was suggested before by Graham and Rhodes [Gra75], but does not appear to have ever been implemented.

9.4 Recording Repairs

Many error recovery systems test a variety of potential recoveries before deciding which recovery to apply. Therefore, it must be possible to keep track of which potential recovery is the best of those tested thus far. The Pascal auditor uses an *ad hoc* encoding for each type of recovery. A better alternative would be to use a generalized representation that could encode any potential repair. Note that any repair can be represented as some combination of deletions and insertions. Furthermore, the repair algorithms developed until now can affect only a few positions within the input text. Therefore, a general repair could be represented by a small vector whose elements represent some sequence of insertions and deletions.

9.5 The Spelling Matcher

The spelling matcher has proven to be a valuable component of the Pascal auditor. While only a small percentage of recoveries are influenced by the spelling matcher, the incorrect recoveries that would be generated for those cases if the spelling matcher were not used would seem outrageous to the naive user. For example, in one program in the Ripley-Druseikis sample, the keyword **function** is misspelled as "funtion." Normally, the Pascal auditor will replace the identifier "funtion" with the keyword **function**. However, if the spelling matcher is disabled, the Pascal auditor will replace "funtion" with the keyword **procedure**.

The spelling matcher takes two input parameters: a source string and a target string. Both the source string and the target string are assumed to end with a null byte. The spelling matcher determines if the source string is a close enough match to the target string that it could safely be assumed to be a misspelling of the target string. The Pascal auditor uses the spelling matcher to decide if an identifier might be considered a misspelling of a keyword.

The spelling matcher is much simpler than the spelling correctors used in compilers such as CUPL[Mor70]. A spelling corrector takes a string and trys to find the keyword or defined identifier which most closely matches the string. Thus, a spelling corrector must try to match the string with every keyword and every identifier in the symbol table. The Pascal auditor does not use a spelling corrector because the time and space overheads associated with the spelling correctors that appear in the literature were felt to be too great.

The algorithm used by the spelling matcher is shown in Figure 9.1. The type *string* is a 1-indexed array of characters. The strings are converted to lowercase because ANSI Pascal does not differentiate on the basis of case. The conversion for the target string is a wasted operation for the current Pascal auditor; the target strings passed to the spelling matcher never contain uppercase characters. The special cases are provided to allow for common substitutions for keywords that are not caught by the general algorithm. The Pascal auditor recognizes four special cases: "constant" matches the keyword **const**, "over" matches the keyword **div**, "go" matches the keyword **goto**, and "proc" matches the keyword **procedure**.

The loop that forms the bulk of the algorithm counts the number of changes that must be made to the source string to make it match the target string. If that number is less than or equal to a limiting value, the two strings are considered close matches and so true is returned; otherwise, the algorithm returns false. The limiting value is set to one-third the length of the source string. The loop scans through the source and target strings checking if the corresponding characters are the same. Whenever it finds a mismatch, it performs a sequence of tests to decide how to continue. It tests for permutations, insertions, substitutions, and deletions, in that order. The order in which the tests are done is significant. The test for permutations must precede all other tests because the other tests can mask the presence of a permutation.

```
function spell(source, target: string): Boolean;
begin
    convert all uppercase characters in the source string to lowercase;
    convert all uppercase characters in the target string to lowercase;

    if the arguments form a special case then
        return true;

    source_length := the length of the source string;
    target_length := the length of the target string;
    limit := source_length div 3;

    number_of changes := 0;
    i := 1;
    j := 1;
    while (i  source_length) and (j  target_length) do
        if source[i] = target[j] then
            begin  i := i + 1;  j := j + 1  end
        else
        begin
            if number_of_changes = limit then
                return false;

            number_of_changes := number_of_changes + 1;
            if (source[i] = target[j + 1]) and (source[i + 1] = target[j]) then
                begin  i := i + 2;  j := j + 2  end
            else if source[i + 1] = target[j] then
                begin  i := i + 2;  j := j + 1  end
            else if source[i + 1] = target[j + 1] then
                begin  i := i + 2;  j := j + 2  end
            else
                j := j + 1
        end

    if number_of_changes + abs(i - target_length) ≤ limit then
        return true
    else
        return false
end
```

Figure 9.1 The spelling matcher

10
Future Work

Experience with Bison and the Pascal auditor has revealed many lines for further research. Ideas for improving the error recovery techniques described in earlier chapters are discussed in those chapters. This chapter describes ideas for future work that lie outside the purview of earlier chapters.

10.1 New Test Suites for Error Recovery

The Ripley-Druseikis sample of erroneous Pascal programs [RD78] has been a valuable contribution to research in error recovery. By providing a standard set of test examples, it has made meaningful comparisons of diverse error recovery systems possible. However, the Ripley-Druseikis sample has become dated.

The programs on which the Ripley-Druseikis sample was based were gathered at the University of Arizona computing center in the mid 1970's. At the time, that center was mainly a Fortran shop. Keypunches and batch processing were the rule of the day. Changes in programming environments over the years have, to some extent, altered the types of errors people make. For example, the Ripley-Druseikis sample includes programs where an error was corrected, but a copy of the line containing the error was left in the program. This type of error is common in a punched card environment (the user simply forgets to throw a bad card away). It is not common when programs are created using an interactive text editor.

There are many types of errors that are not represented in the Ripley-Druseikis sample. Several features of Pascal are not used in any of the programs. No record type specifiers, case-statements, or with-statements appear in the sample. Presumably, the number of programs that contained those features fell below the threshold needed to be included. Also, since the sample was created for the analysis of syntactic errors, semantic errors, even those that result from syntactic causes, are not represented.

Since the erroneous programs were gathered at a university, it may be inferred that the sample is representative of the types of errors made by student programmers. It seems unlikely that production programmers make the same types of errors as students; however, no evidence has been gathered to support that conjecture. A statistically weighted sample of erroneous programs written by production programmers could provide the evidence needed to assess that conjecture's validity.

The fact that the only standard suite of erroneous programs is a sample of Pascal programs has led those who study error recovery to concentrate on error recovery for Pascal. Indeed, the fact that the Ripley-Druseikis sample was composed of Pascal programs was the main reason why the auditor used to test the error recovery techniques described herein was written for Pascal. The effectiveness of an error recovery technique may differ for various languages. Error samples for other languages could serve as vehicles for showing that an error recovery technique is robust.

Creating new error samples is a nontrivial task. To obtain a statistically valid sample, thousands, perhaps tens of thousands, of erroneous programs must be gathered. Each program must be inspected to determine the nature of the errors contained in it. Only then can a representative sample be extracted. Just obtaining an unbiased collection of erroneous programs can be difficult. A compiler could be modified to save copies of programs containing errors. However, because programmers are likely to compile several erroneous versions of a program before eliminating all detectable errors, a sample gathered in that way is likely to be biased.

10.2 Error Productions

As was noted in the introduction, error productions extend the syntax of the language to be analyzed. Error productions can be used to handle errors that could not otherwise be handled well. Examples of uses for error productions are given by Fischer and Mauney [FM80]. Error productions must be provided by the compiler writer. Obtaining good results from error productions requires a sharp sense of which error productions should be provided. As a rule, error productions should be used to relax nonintuitive syntactic restrictions.

Error productions are useful, but they are also troublesome. Error productions may cause an error recovery system to mishandle some errors that they would otherwise handle well. For example, when Berkeley Pascal analyzes the declaration

const *kp1* $= k + 1$;

it produces

```
        4  const kp1 = k + 1;
    E --------------------^--- Expected ';'
    e --------------------^--- Replaced '+' with a keyword label
```

The apparent error is that an expression has been used where a constant is required. The poor recovery is the result of error productions: one missing, and one present. If the grammar for Berkeley Pascal contained error productions permitting expressions to appear in most contexts where constants are allowed, as does the Pascal auditor, the error would have been handled well. On the other hand, Berkeley Pascal does contain error productions allowing declarations to appear in any order. A stricter grammar would not permit a label declaration to follow a constant declaration, and so it would be impossible to substitute the keyword **label** for the operator '+'.

Error productions may cause a grammar that was acceptable to a parser generator to cease being acceptable. Recall that in Section 8.3, it was mentioned that the bounds of a subrange could not be allowed to be expressions. If a parenthesized expression is allowed as the bound of a subrange, then the bound of a subrange can be syntactically equivalent to an enumerated type. For example, in the declaration

type t = (*red*)..*green*;

it is impossible for the parser to determine that *red* is an expression and not the name of an enumeration constant until the token '..' is read. While it is possible to write a grammar that allows delaying the decision of whether to reduce *red* to an expression or an enumeration constant until after the two following tokens have been read, the natural grammar for that construction is LR(2).

Another problem of error productions is illustrated by an idea suggested by Fischer and Mauney [FM80]. Consider the code fragment

$$a[i] \;=\; a[j] \;\textbf{then}\; p(i, j);$$

where a is an array variable. The likely error is that the keyword **if** has been deleted from the start of the line. However, the error will not be detected until the equals sign has been read. By that time, the parser will have performed shifts and perhaps reductions that will preclude inserting the keyword **if** at the start of the line. Therefore, unless the recovery algorithm can back up the parse, the best repair possible is to replace the operator '=' with the symbol ':=' and the keyword **then** with the symbol ';'. Fischer and Mauney suggest using error productions to deal with this type of error. They advocate adding error productions that permit parsing an if-statement without a leading **if**. As they note in their article, those error productions will permit the previous error to be handled well. What they fail to note is that the same productions will prevent a more common error from being handled well. For example, consider the statement

$$a[i] \;=\; a[j];$$

The apparent error this time is that the symbol '=' appears where the symbol ':=' was intended. This error is easy to handle if the parser recognizes the error before shifting over the equals sign. However, if the error productions suggested by Fischer and Mauney are used, an expression will be allowed to appear as the head of a statement. Therefore, the error will go undetected until the semicolon is read. Hence, an error recovery algorithm that cannot back up the parse will be unable to replace the symbol '=' with the symbol ':='.

Thus, error productions are shown to be a valuable but flawed tool for error recovery. There may be ways of minimizing the harmful effects of error productions without reducing their usefulness. Error productions could be distinguished from normal productions. A parser generator might accept a strict grammar for a language together with a set of error productions. The error productions could either augment the rules of the strict grammar or supplant them. For example, the rules for the if-statement missing the leading **if** would augment the normal rules for statements, whereas the rules substituting expressions for constants would supplant the strict rules for constants. A parser for the strict grammar could be generated along with tables indicating how the parser would be different if the error productions were used. During normal compilation, the parser would recognize the strict form of the language. Thus, syntax errors would be detected as soon as possible. When an error was detected, the parser would be backed up to states in the parse stack that the parser generator indicates would be different if the error productions had been used. If parsing is able to continue without error until the full construct described by the error productions is recognized, normal parsing could then be resumed. Otherwise, the rest of the error recovery machinery would be invoked. The recovery algorithm could consider using error productions as part of a recovery, but be biased against such recoveries. The problem of the mechanism just described is its cost. The ability to back up the parse to those states that would be different if the error productions were used is not quite as expensive as a general backup facility, but it comes close. Further, the information that would have to be stored to make the system work could require more table space than the parser itself.

10.3 Improving the Parser Generator

The parser generator Bison was designed to be a research tool. The parsers created by Bison incorporate only a few of the error handling features described herein, namely, LR(2) checking for syntax errors, semantic guards, and the location stack. The remaining error handling capabilities must be provided by the user. When a Bison parser detects an error, it restores the parse stack if necessary and then calls a routine named *yyrecover*. *Yyrecover* must be supplied by the user. This arrangement was adopted because the error recovery routines were in a state of flux. The recovery routines would have taken much longer to develop had it been necessary to modify Bison to test the effects of a change.

If Bison were to be made into a production parser generator, it should generate the error recovery routines automatically. Most compiler writers cannot be expected to know or care how error recovery is done, so long as it is done well. The recovery algorithm created in this work could easily be incorporated into a parser generator. The compiler writer still must supply some information to the recovery system. The cost functions *Icost*, *Dcost*, and *Rcost* are language dependent and so must be provided by the compiler writer. The compiler writer must also designate which tokens should be regarded as brackets and which should be regarded as separators.

A parser generator for production use should possess capabilities that were not included in the current version of Bison. Aho, Johnson, and Ullman [AJU75] showed that smaller and faster parsers can be created for practical languages if ambiguous grammars are used. Bison incorporates some of their suggestions, but not all. In particular, their system for using operator precedence to resolve ambiguities has not been implemented. For Pascal, that feature is not important because there are few levels of precedence. Some other languages, such as C [KR78], have many levels of precedence. The parsers for those languages can be made more efficient by using ambiguous grammars and precedence declarations.

Some texts on compiler writing, such as [AU77], indicate that LALR(1) grammars are suitable for describing the syntax of almost all programming languages. Experience gained while writing the Pascal grammar used by the Pascal auditor contradicts that notion. Even ignoring factors affecting error recovery, the rules describing declarations and record type specifiers must be carefully constructed to make the grammar LALR(1). However, both of those features can easily be described in an LALR(2) grammar. Other languages also contain constructs that can be described more naturally with LALR(2) grammars than with LALR(1) grammars. Wetherell [Wet81] describes such a construct in Ada. Extending Bison to handle LALR(2) grammars should not be difficult. Most of the information needed to compute the LALR(2) lookahead sets is present in the DeRemer-Pennello algorithm for computing LALR(1) lookahead sets [DP82].

Using a contorted grammar can hurt the quality of error recovery. Automatic error recovery techniques are heavily influenced by the grammar. Therefore, the structure of a grammar should correspond as closely as possible to the programmer's view of the syntactic structure of the language. When the grammar deviates from that conceptual syntactic structure, recoveries that are not intuitively appealing may result. In the Pascal auditor, a few recoveries suffer because the syntax of declarations does not match what a programmer would regard as the syntactic structure of declarations. If an LALR(2) grammar for declarations could have been used, those problems would not have arisen. On the other hand, the tortuous rules needed to describe record type specifiers do not appear to affect the recoveries chosen.

10.4 Enhancing the Local Recovery Algorithm

One reason the Pascal auditor generally diagnoses errors more accurately than does Berkeley Pascal is that it considers more potential repairs. Furthermore, in almost every instance where Berkeley Pascal or the Burke-Fisher system outperforms the Pascal auditor, it is because they find a repair that was not considered by the Pascal auditor. It seems likely, therefore, that the quality of repairs produced by the Pascal auditor could be improved by having it consider more types of repairs.

An obvious way of increasing the number of repairs considered is to consider combinations of simple repairs. The repairs could even affect different parts of the input text. Some types of errors, particularly bracketing errors, cannot be accurately diagnosed unless such repairs are considered. An early version of the Pascal auditor considered combinations of two simple repairs. That version of the Pascal auditor generally found better repairs than the current one. However, the time needed to select a recovery was too great. The current repair algorithm typically considers a few hundred potential repairs for each error. The earlier version typically tested tens of thousands of potential repairs. That early algorithm screened out cases where a simple repair was obviously best; nonetheless, it was at least two orders of magnitude slower than the current repair algorithm.

Bracket repairs are the remnant of that earlier algorithm. The cost of bracket repairs is significant; the error recovery algorithm spents about 10% of its time considering bracket repairs. The time spent is worth the cost because a significant percentage of errors are diagnosed more accurately. Often where a bracket repair is applied, the recovery algorithm would have been invoked twice if bracket repairs were not done, once for the opening bracket and once for the closing bracket.

Thus, combinations of repairs have been found to be too expensive to apply in general, but have been useful in a special case. It seems unlikely that a recovery algorithm that allows general combinations of repairs can be made practical. However, there may be other special combinations of repairs that are worth considering.

The error recovery algorithm could both save time and produce better quality repairs if the length of each forward move could be varied appropriately. The current limit on the length of a forward move is twelve tokens. Most errors can be diagnosed accurately without looking at more than nine tokens. Therefore, the recovery algorithm typically wastes time performing unnecessarily long forward moves. On the other hand, bad repairs are sometimes selected because the repair algorithm does not look ahead far enough. Suppose a program contains a line of the form

$$\textbf{if } i \;\; j + \cdots + k \textbf{ then}$$

where "..." represents an arithmetic subexpression, and i, j, and k are integer variables. An error will be detected between i and j since identifiers cannot be adjacent in Pascal. The Pascal auditor repairs the error by inserting an operator between i and j. If the subexpression is less than 8 tokens long, the local recovery algorithm will read the token

then while testing potential repairs. Therefore, it will know that the expression must be of type Boolean, and so it will insert the relational operator '='. Otherwise, it would insert the arithmetic operator '+'. Inserting the relational operator is clearly the better repair. However, in other instances, the expression could be followed by a relational operator and another arithmetic expression. In such a case, inserting the arithmetic operator would be better. The recovery algorithm cannot tell which case applies unless the forward move reaches the token **then**.

The current repair algorithm is naive in its use of semantic costs. Experience has shown that the range of costs is unnecessarily large, while the range of effects is undesirably narrow. The costs assigned to semantic errors have little effect on the choice of repairs. Recall that the semantic cost of a potential repair is the sum of the costs returned by the semantic guards during the forward move done to evaluate the repair. The repair algorithm always chooses a repair whose semantic cost is zero if any such repair is found. For almost all of the programs in the sample of erroneous programs used to measure the performance of the Pascal auditor, there was at least one potential repair whose semantic cost was zero. Therefore, the costs assigned to semantic errors usually did not matter since if the semantic cost of a potential repair were little as one, it usually was great enough to preclude the choice of that repair. For this reason, Schmauch's simple scheme for semantics-directed error recovery (see Chapter 3) should usually be as effective as that used by the Pascal auditor.

Semantic costs not only influence the choice of repairs, but can absolutely block the choice of a particular repair. If the semantic cost of a potential repair exceeds some limit value, the repair is automatically rejected. In retrospect, this mechanism could prevent some errors that would otherwise be easy to repair from being repaired. Suppose that a syntax error is closely followed by an unrelated semantic error whose semantic cost exceeds the limit value. Then all possible repairs of the syntax error will be rejected because of that unrelated error. No examples of this performance have yet been observed, but it could happen.

The integration of semantic and syntactic costs could be handled better. It is not clear that potential repairs whose semantic cost is zero should always be favored over repairs whose semantic cost is positive. Instead of having a wide range of semantic costs representing the severity of the error, it might be better to group the errors into a small number of classes, each of which is treated differently. There might be four classes of semantic errors: trivial, normal, severe, and intolerable. A trivial semantic error would have the effect of adding a small amount to the syntactic cost of a repair. A repair for which only trivial semantic errors are found while testing the repair would be favored over all repairs for which more serious semantic errors are found, but would not necessarily be rejected if a repair for which no semantic errors are detected is found. Normal errors would be treated much the same as semantic errors are currently treated. A repair that leads to a severe semantic error might not be permitted if the associated syntactic cost were too great. Repairs that lead to intolerable errors would not be permitted at all. For example, in Pascal, a scalar variable cannot be followed by a left bracket. Therefore, inserting a left bracket following a scalar variable might lead to an intolerable semantic error. In addition to preventing some bad recoveries, treating some semantic errors as intolerable errors may improve efficiency by causing forward moves for bad repairs to be ended sooner than they might otherwise be ended.

It should be noted that the current system of using semantic costs works well. Although the ideas mentioned above may sometimes lead to better repairs, the current semantics-directed repair algorithm produces good recoveries in most cases. The suggestions made above are the result of reflections on ways to improve the algorithm rather than any demonstrated need for improvement.

Multiple errors in a single context often lead to inferior recoveries. For example, there are two errors in the statement

$$i = j + k)\ \textbf{then}\ \textbf{goto}\ 10$$

The keyword **if** is missing from the start of the line, and an extra right parenthesis appears between k and the keyword **then**. The first error is detected as soon as the token '=' is read. At that point, the error repair algorithm is able to insert the keyword **if** at the start of the line. However, because of the second error, the forward move never reaches the keyword **then**. Therefore, the best repair is rejected in favor of a less desirable repair.

The problems posed by multiple errors can easily be solved if efficiency is not a consideration. For each viable repair for the first error, the error repair algorithm could find the best repair for the second error after patching the first error. The best combination of repairs could be chosen based on the total cost of both repairs. Further research might reveal an efficient technique that produces similar results.

10.5 Other Languages

Perhaps the most important unanswered question about the error recovery techniques described herein is whether will they carry over to languages other than Pascal. Pascal is an almost ideal vehicle for error recovery. Its baroque syntax and restrictive semantic rules often lead to errors that are easily repaired. Moreover, the language is highly redundant. Therefore, a potential repair's viability can almost always be determined by examining the surrounding context.

Shebanow is currently implementing a front end for a C compiler that incorporates advanced versions of the error recovery techniques described herein. Error recovery for C appears to be more difficult than for Pascal. C is much less redundant than Pascal. For example, in Pascal, if the keyword **if** were missing from the start of a conditional statement, the subsequent keyword **then** would signal its absence. Now consider the following erroneous code fragment from a C program

$$(i == j)\ key = table[i];$$

The error in this example is that the keyword **if** is missing at the start of the line. Because C is an expression language, no error is detected until the identifier *key* is read. At that point, the most likely repair would be to insert a semicolon (';') before *key*. Even if the recovery algorithm did decide that a keyword was missing from the start of the line, it would have to make an arbitrary choice between inserting **if** or **while**.

The use of semantics during error recovery probably will not prove as advantageous for C as it did for Pascal. The type rules of C are much less strict than those of Pascal. Therefore, many semantic errors are undetectable. On the other hand, bracket repair should be even more effective in C than it was in Pascal. C contains more bracketed constructs than does Pascal and more types of brackets.

Ada [DoD83] seems well suited for the error recovery techniques described herein. Certain of Ada's constructs cannot be identified by syntactic information alone. Therefore, semantics-directed error recovery may prove essential to avoiding frequent misdiagnosis of errors. Further, Ada is highly redundant; even more so than Pascal.

Implementing semantic-directed error recovery for Ada is complicated by Ada's rules for resolving overloading. The type of a component of an expression in general cannot be determined until after the entire expression has been parsed. Therefore, semantic errors will sometimes go undetected until long after the point in the text where the error was made. Therefore, an error recovery system for Ada may need to be able to backtrack the analysis of expressions.

Semantics-directed error recovery could be applied to languages that do not require definition before use; however, the cost of doing so may be high. Compilers for languages that allow use before definition are usually multi-pass compilers. Implementing semantics-directed error recovery for such a language may require an additional pass that precedes parsing. That pass would determine the block structure of the program and process all recognizable declarations, but it would not parse statements and expressions. Koster [Kos73] has reported some work along these lines. Semantics-directed error recovery can then be done while the program text is parsed, since the necessary semantic information will be available.

An implementation such as that outlined above suffers from two problems. First, the global bracketing structure of the program will have to be parsed without the benefit of semantics-directed error recovery. This is not a great disadvantage, since, in most languages, the types of semantic information available during semantics-directed error recovery would have little effect on the choice of recoveries. The other problem is that a declaration may be missed by the earlier pass because of an error. The error may not be discovered until after other errors have been repaired based on faulty information. The error recovery system would then have to back out of any actions done as a result of those repairs. Any scheme for providing that type of capability is likely to be too slow to be practical.

11 Conclusions

The major result of this work was the development of practical techniques for applying general static semantic information to assist in recovering from syntactic errors. This result was achieved by extending the Graham-Haley-Joy error recovery algorithm to take semantic costs into account when selecting a repair.

The main obstacle to the creation of a system for applying semantics to error recovery was the need to be able to reverse the effects of semantic actions. To be able to determine the semantic cost of a repair, semantic checking must be done in tandem with parsing. However, semantic checking involves performing the semantic actions associated with the syntactic constructs being recognized. Since performing semantic actions affects the state of compilation, the effects of semantic actions performed while testing potential repairs that are rejected must be negated so that compilation can continue normally after recovering from an error.

No practical solution to the problem of negating the effects of semantic actions performed by conventional compilers was found. Therefore, it was necessary to formulate a restricted model of compilation that was suited to undoing those effects. It was found that the LL- and LR-attributed grammars constituted just such a model. However, currently, compilers based on those types of attribute grammars are too slow to be practical even for normal compilation. Therefore, a new model of compilation was formulated. The new model consists of synthesized attributes together with a symbol table that allows the effects of symbol table operations to be undone. Two symbol table organizations suited for use in such a model were developed.

Another contribution of this work concerned methods for avoiding the deleterious effects of reductions performed because of erroneous input. The methods included general backtracking, suppressing default reductions, LR(1) pretesting, stack restoration, and limited backtracking. LR(1) pretesting and limited backtracking are new techniques developed during this work. General backtracking, suppressing default reductions, and stack restoration were first suggested by others. Comparisons of the various methods showed that limited backtracking was the best method for use with semantic-directed error recovery.

The final contribution of this work was a new panic mode recovery technique for use with LR parsers. The new technique is largely a synthesis of existing techniques. It is essentially an adaptation of Hartmann's panic mode technique for recursive descent parsers [Har77]. Hartmann's technique depends on knowing the set of nonterminal symbols currently involved in the derivation being produced. A close approximation to that information for bottom-up parsers is provided by Sippu and Soisalon-Soininen's concept of a feasible reduction goal [SS83]. The new technique incorporates the concept of feasible reduction goals into a panic mode algorithm for LR parsers that is closely related to Hartmann's technique.

The goal of this work was to develop better techniques for error diagnosis. Comparisons with two of the best existing error recovery systems, Berkeley Pascal [GHJ79] and the Burke-Fisher system [BF82], show that that goal has been achieved.

Further, timings demonstrate that the execution time overheads associated with the new techniques are reasonable. The Pascal auditor analyzes error-free programs at about the same speed as Berkeley Pascal. On the other hand, the Pascal auditor takes longer to recover from an error than does Berkeley Pascal. On average, it is from two to three times slower. Nonetheless, the new system is fast enough to be practical. When running on a VAX-11/780, the Pascal auditor typically requires less than one-tenth of a second to recover from an error.

The main drawbacks of the new error recovery techniques are their space requirements and the difficulty of applying them. The code and tables for the Pascal auditor occupy about 12% more space than the translator for the Berkeley Pascal interpreter. If the Pascal auditor were extended to generate code similar to that produced by the translator for Berkeley Pascal, it would be at least 50% larger than that translator. Also, the new symbol table organization requires more space than would be needed by a conventional compiler. The greater space requirements are probably irrelevant for major computing systems. They may, however, be an obstacle to use of the new techniques in compilers for computers with limited address spaces.

The difficulty of applying the new techniques is likely to be an impediment to their use. When a conventional compiler detects an error during semantic analysis, it simply produces an error message. The semantics-directed error repair algorithm requires the compiler writer to provide additional information about semantic errors. The semantic routines must indicate which errors should cause the error recovery algorithm to be invoked, and they must assign costs to each error. To fulfill these requirements, the semantic routines must do a more detailed error analysis than is done by the semantic routines of conventional compilers. Further, the restrictions prohibiting semantic actions from altering existing semantic attributes or global variables outside of the symbol table sometimes force the compiler writer to use algorithms or data structures for semantic analysis that he would not normally choose.

The techniques for applying general static semantic information to syntactic error recovery are the culmination of one line of research into error recovery. The error recovery systems of Feyock and Lazarus [FL76] and Graham, Haley, and Joy [GHJ79] were able to improve on earlier systems by using some semantic data to help detect and recover from syntax errors. The error recovery techniques described herein carry that idea to its practical limits. All static semantic information that would normally be produced by a compiler is used by the new techniques. The techniques could be extended to find and make use of information not needed during normal compilation, and better recoveries could sometimes be done using that additional data. However, the proportion of cases where such information would be useful is small, and so the time spent gathering it will usually be wasted.

Appendix A
The Grammar for the Pascal Auditor

This appendix contains a listing of the grammar used in the implementation of the Pascal auditor. For ease of reference, each rule is labeled by a distinct rule number. The names of terminal symbols are written in uppercase, and the names of nonterminal symbols appear in lower case. Some of the reasons for using this particular grammar are given below.

The names of most terminal symbols clearly indicate which symbols they denote. For example, each name that denotes a keyword has the same spelling as the keyword that it denotes. The meanings of a few names are not as obvious. Those names and their meanings are as follows:

ID	an identifier	
INT	an unsigned integer constant	
REAL	an unsigned real constant	
STRING	a character string constant	
DOT	a period	'.'
UPTO	the range symbol	'..'
ARROW	a caret	'^'
BECOMES	the assignment symbol	':='
LPAR	a left parenthesis	'('
RPAR	a right parenthesis	')'
LBRAK	a left bracket	'['
RBRAK	a right bracket	']'

In addition, there are two special tokens, ERROR and FORWARD, which are not a part of the language. The Pascal auditor's lexical analyzer never returns either of those tokens. ERROR is the error token recognized by the Pascal auditor's panic mode algorithm (see Section 7.3). FORWARD is a pseudo-keyword. In ANSI Pascal [ANS83], the symbol "forward" is an identifier. However, it is distinguished from other identifiers in that it may appear as a forward directive. The special token FORWARD is used to allow the spelling matcher to replace an identifier with a forward directive. The special token is needed because the spelling matcher tries to match identifiers with keywords but does not try to match identifiers with other identifiers.

Rules 2, 129, and 141 were included to prevent the panic mode algorithm from performing certain bad recoveries. Note that these rules are simple chain rules whose elimination would in no way affect the language defined by the grammar. However, before these rules were added, whenever the parser encountered a declaration with the statement part of a block, the panic mode algorithm would discard the preceding portion of the statement part of the block and back up to the point where it could begin processing new declarations. This action would sometimes throw the parser off the track so that many spurious errors would be detected, and it would sometimes cause the nesting level of the symbol table to be altered. It is interesting to note that Berkeley Pascal suffers from similar problems.

The rules defining the nonterminal symbol *newtype* contain some notable features. Rule 16 is the rule where the nonterminal symbol *konst* which defines a constant could not be replaced with the nonterminal symbol *expr* because an LR(1) conflict would result (see Sections 8.3 and 10.2). The nonterminal symbol *closer* that appears on the rhs of rule 18 derives the empty string. It is used to close the scope that is opened when the start of a record type specifier is recognized (see rule 36). Rules 19 and 20 are error productions that allow general types to appear in contexts where the strict definition of the language permits only restricted subclasses of types. Rule 22 is also an error production. It permits a general type specifier to appear following a caret. ANSI Pascal requires the type specifier following a caret to be a type identifier.

Rules 36 through 55 define the syntax of a record type specifier. The tortuous definition given was not chosen because of any considerations regarding error recovery. Rather, it was the simplest definition found that did not cause an LALR(1) conflict. A simpler definition could have been used if either ANSI Pascal had defined the syntax of record type specifiers slightly differently or if an LALR(2) parser generator had been available. Rule 55 is an error production that permits a general type specifier to appear where the strict language would require a type identifier.

Rules 74 and 91 - 101 define the syntax of a parameterized procedure statement. The rules had to be factored in this way to permit the declaration of the procedure identifier to be percolated up to the parameter expressions so that their types could be checked as they were recognized. If the semantic routines of the Pascal auditor were allowed to use inherited attributes, a more natural syntax could have been used (see Section 5.3).

Rules 125 and 126 define control symbols for changing the nesting level. The semantic routine for the nonterminal symbol *opener* increases the nesting level by one. The semantic routine for the nonterminal symbol *closer* pops the current scope.

Rules 133 is an error production that allows a return type to be specified in a procedure header. The nonterminal symbol *prcerr* is a control symbol whose only function is to signal that the error recovery algorithm should be invoked. The reason the error recovery algorithm is invoked before the error production can be applied is to allow for the possibility that the tokens that appeared to form a return type specification for the procedure are really the result of some other error.

Rules 202 through 217 have been factored in a way that allows information about the name that appears at the left of an *lval* to be made available to the semantic routines for the components of the *lval*. Again, if inherited attributes could have been used, this syntax could have been simplified.

```
 1   program  :  gbldcls block DOT


 2   gbldcls  :  pgmdcls


 3   pgmdcls  :  pgmhead
 4            |  pgmdcls dcl


 5   pgmhead  :  PROGRAM ID propopt SEMICOLON
 6            |  PROGRAM ERROR


 7   propopt  :
 8            |  LPAR propars RPAR


 9   propars  :  ID
10            |  propars COMMA ID


11   type     :  typname
12            |  newtype
13            |  ERROR


14   typname  :  ID


15   newtype  :  LPAR vnames RPAR
16            |  konst UPTO konst
17            |  packopt ARRAY LBRAK indxtys RBRAK OF type
18            |  record closer
19            |  packopt SET OF type
20            |  packopt FILE OF type
21            |  ARROW ID
22            |  ARROW newtype


23   vnames   :  ID
24            |  vnames COMMA ID


25   konst    :  INT
26            |  STRING
27            |  name
28            |  PLUS INT
29            |  MINUS INT
30            |  PLUS name
31            |  MINUS name
```

```
32    packopt  :
33             |  PACKED


34    indxtys  :  type
35             |  indxtys COMMA type


36    record   :  packopt RECORD opener fldlist END


37    fldlist  :  fixdhead
38             |  fixdpart
39             |  varipart
40             |  variend


41    fixdhead :
42             |  fixdpart SEMICOLON


43    fixdpart :  fixdhead fnames COLON type


44    fnames   :  ID
45             |  fnames COMMA ID


46    varihead :  fixdhead CASE selector OF
47             |  variend


48    varilist :  varihead konst
49             |  varilist COMMA konst


50    varipart :  varilist COLON LPAR fldlist RPAR


51    variend  :  varipart SEMICOLON


52    selector :  seltype
53             |  ID COLON seltype


54    seltype  :  typname
55             |  newtype


56    block    :  BEGIN stmts END


57    name     :  ID
```

```
58   dcl      :  labeldcl
59            |  constdcl
60            |  typedcl
61            |  vardcl
62            |  procdcl
63            |  fnctdcl
64            |  ERROR


65   stmts    :  stmt
66            |  stmts SEMICOLON stmt


67   stmt     :  labels ustat
68            |  ustat


69   labels   :  INT COLON
70            |  labels INT COLON


71   ustat    :
72            |  dest BECOMES expr
73            |  name
74            |  call0 RPAR
75            |  erropt GOTO INT
76            |  erropt BEGIN stmts END
77            |  erropt IF pred THEN stmt
78            |  erropt IF pred THEN stmt ELSE stmt
79            |  erropt CASE caselist END
80            |  erropt CASE caselist SEMICOLON END
81            |  erropt REPEAT stmts UNTIL pred
82            |  erropt WHILE pred DO stmt
83            |  erropt FOR forvar BECOMES forexpr TO forexpr
                 DO stmt
84            |  erropt FOR forvar BECOMES forexpr DOWNTO forexpr
                 DO stmt
85            |  erropt WITH opener withlist DO stmt closer
86            |  ERROR


87   erropt   :
88            |  ERROR


89   dest     :  name
90            |  lval


91   call0    :  call2
92            |  call4
93            |  call6
```

```
94     call1     :  prname LPAR
95               |  call0 COMMA


96     call2     :  call1 expr


97     call3     :  call2


98     call4     :  call3 COLON expr


99     call5     :  call4


100    call6     :  call5 COLON expr


101    prname    :  name


102    pred      :  expr


103    caselist:    casehead COLON stmt


104    casehead:    expr OF konst
105              |  caselist SEMICOLON konst
106              |  casehead COMMA konst


107    forvar    :  name


108    forexpr   :  expr


109    withlist:    recval
110              |  withlist COMMA recval


111    labeldcl:    LABEL lablist SEMICOLON


112    lablist   :  INT
113              |  lablist COMMA INT


114    constdcl:    CONST consteqv SEMICOLON
115              |  constdcl consteqv SEMICOLON
```

```
116     consteqv:  ID EQ expr


117     typedcl :  TYPE typeeqv SEMICOLON
118             |  typedcl typeeqv SEMICOLON


119     typeeqv :  ID EQ type


120     vardcl  :  VAR varcore SEMICOLON
121             |  vardcl varcore SEMICOLON


122     varcore :  idlist COLON type


123     idlist  :  ID
124             |  idlist COMMA ID


125     opener  :


126     closer  :


127     procdcl :  plcdcls block SEMICOLON closer
128             |  prchead SEMICOLON dirctiv SEMICOLON


129     plcdcls :  prcdcls


130     prcdcls :  prchead SEMICOLON
131             |  prcdcls dcl


132     prchead :  PROCEDURE pid paropt
133             |  PROCEDURE pid paropt prcerr prcret
134             |  PROCEDURE ERROR
135             |  PROCEDURE pid ERROR


136     prcerr  :


137     prcret  :  COLON type


138     pid     :  ID
```

```
139     fnctdcl  :  flcdcls block SEMICOLON closer
140              |  fnchead SEMICOLON dirctiv SEMICOLON


141     flcdcls  :  fncdcls


142     fncdcls  :  fnchead SEMICOLON
143              |  FUNCTION fid SEMICOLON
144              |  fncdcls dcl


145     fnchead  :  FUNCTION fid paropt COLON restype
146              |  FUNCTION fid parpack
147              |  FUNCTION ERROR
148              |  FUNCTION fid ERROR


149     fid      :  ID


150     restype  :  typname
151              |  newtype


152     dirctiv  :  ID
153              |  FORWARD


154     paropt   :
155              |  parpack


156     parpack  :  LPAR opener pars RPAR closer
157              |  LPAR opener ERROR closer RPAR


158     pars     :  par
159              |  pars SEMICOLON par


160     par      :  parids COLON partype
161              |  VAR parids COLON partype
162              |  FUNCTION ffid paropt COLON restype
163              |  PROCEDURE fpid paropt


164     parids   :  ID
165              |  parids COMMA ID


166     partype  :  typname
167              |  newtype
```

```
168     ffid     :  ID

169     fpid     :  ID

170     expr     :  sexpr
171              |  sexpr EQ sexpr
172              |  sexpr NE sexpr
173              |  sexpr LT sexpr
174              |  sexpr GT sexpr
175              |  sexpr LE sexpr
176              |  sexpr GE sexpr
177              |  sexpr IN sexpr
178              |  ERROR

179     sexpr    :  term
180              |  PLUS term
181              |  MINUS term
182              |  sexpr PLUS term
183              |  sexpr MINUS term
184              |  sexpr OR term

185     term     :  factor
186              |  term STAR factor
187              |  term SLASH factor
188              |  term DIV factor
189              |  term MOD factor
190              |  term AND factor

191     factor   :  name
192              |  lval
193              |  INT
194              |  REAL
195              |  STRING
196              |  NIL
197              |  fncall
198              |  LBRAK RBRAK
199              |  LBRAK members RBRAK
200              |  LPAR expr RPAR
201              |  NOT factor

202     lval     :  recval DOT ID
203              |  ptrval ARROW
204              |  subhead RBRAK

205     recval   :  name
206              |  lval
```

```
207    ptrval   :  name
208             |  lval

209    subhead  :  arrval LBRAK expr
210             |  subhead COMMA expr

211    arrval   :  name
212             |  lval

213    fncall   :  fnpart RPAR

214    fnpart   :  fnhead expr

215    fnhead   :  fnname LPAR
216             |  fnpart COMMA

217    fnname   :  name

218    members  :  expr
219             |  expr UPTO expr
220             |  members COMMA expr
221             |  members COMMA expr UPTO expr
```

Appendix B

Recoveries Produced with and without Semantics

Although comparing the recoveries produced by the Pascal auditor with those produced by Berkeley Pascal and the Burke-Fisher system yields impressive results, those comparisons do not clearly demonstrate the advantages of semantics-directed error recovery. The differences in the recoveries produced by those systems are often due to factors that are not directly related to the use of semantics. The best test of the benefits of using semantic data to aid in error recovery lies in comparing the recoveries produced by the same error recovery system with and without semantics. To this end, the Pascal auditor has been used to produce recoveries for the Ripley-Druseikis test suite both with semantics enabled and with semantics disabled. Different recoveries were generated for 27 of the 126 programs in the test suite. This appendix contains the listings produced for those programs where the recoveries differed. The numbers used to identify programs are the numbers of their relative positions in the test sample.

```
     1     program p005(input, output);
     2       function getelement(var x: integer); boolean;
                                               ^
***   38:  e - replaced ';' with ':'
     3         var q: integer;
     4       begin
     5         x := 1
     6       end;
     7     begin
     8     end.
```

The listing produced for program 5 with semantics enabled

```
     1     program p005(input, output);
     2       function getelement(var x: integer); boolean;
                                               ^ ^
***   37:  e - missing return type specification
***   40:  e - unknown directive - treated as forward
     3         var q: integer;
               ^
***    5:  e - variable declarations must precede function declarations
     4       begin
     5         x := 1
               ^
***    5:  e - "x" is undefined
     6       end;
             ^
***    3:  e - deleted 'end'
     7     begin
     8     end.
              ^
***    3:  e - inserted 'end' before '.'
```

The listing produced for program 5 with semantics disabled

```
      1   program p011(input, output);
      2     var x, nonprime: integer;
      3         numprime: array [1..10] of integer;
      4   begin
      5     if nonprime = 0 then numprime,x. := numprime(x) + 1;
                                         ^ ^            ^ ^
***  32:  e - replaced ',' with '['
***  34:  e - replaced '.' with ']'
***  47:  e - replaced '(' with '['
***  49:  e - replaced ')' with ']'
      6     x := 1
      7   end.
```

The listing produced for program 11 with semantics enabled

```
      1   program p011(input, output);
      2     var x, nonprime: integer;
      3         numprime: array [1..10] of integer;
      4   begin
      5     if nonprime = 0 then numprime,x. := numprime(x) + 1;
                                         ^ ^    ^
***  32:  e - replaced ',' with '['
***  34:  e - replaced '.' with ']'
***  39:  e - a variable appears where a function was expected
      6     x := 1
      7   end.
```

The listing produced for program 11 with semantics disabled

```
      1    program p023(input, output);
      2      var m, x: integer;
      3          ffact: real;
      4      function fact(n: integer): integer;
      5        begin
      6          fact := 1
      7        end;
      8      function power(k, y: integer): integer;
      9        begin
     10          power := 1
     11        end;
     12    begin
     13      ffact"(power(m, x) * exp(-m)) div (fact(x));
                  ^
***      8: e - replaced '"' with ':='
     14      x := 1
     15    end.
```

The listing produced for program 23 with semantics enabled

```
      1    program p023(input, output);
      2      var m, x: integer;
      3          ffact: real;
      4      function fact(n: integer): integer;
      5        begin
      6          fact := 1
      7        end;
      8      function power(k, y: integer): integer;
      9        begin
     10          power := 1
     11        end;
     12    begin
     13      ffact"(power(m, x) * exp(-m)) div (fact(x));
             ^    ^                      ^              ^
***      3: e - a variable appears where a procedure was expected
***      8: e - deleted '"'
***     31: e - deleted ')'
***     46: e - inserted ')' before ';'
     14      x := 1
     15    end.
```

The listing produced for program 23 with semantics disabled

```
         1     program p024(input, output);
         2       constant pi = 3.14159: real;
                 ^                    ^ ^
 ***      3:   e - replaced 'constant' with 'const'
 ***     24:   e - deleted ':'
 ***     26:   e - deleted 'real'
         3       var x: integer;
         4     begin
         5       x := 1
         6     end.
```

The listing produced for program 24 with semantics enabled

```
         1     program p024(input, output);
         2       constant pi = 3.14159: real;
                 ^             <------^-^-->
 ***      3:   e - replaced 'constant' with 'const'
 ***     17:   e - expression replaced by a constant
 ***     24:   e - replaced ':' with '+'
 ***     26:   e - type name appears where an expression was expected
         3       var x: integer;
         4     begin
         5       x := 1
         6     end.
```

The listing produced for program 24 with semantics disabled

```
       1     program p027(input, output);
       2       const maxrelations = 2;
       3       var x: integer;
       4           prtlrdrdata: array [1..2*maxrelations] of integer;
                                          --
***   31:   e - deleted '*'
***   32:   e - deleted 'maxrelations'
       5     begin
       6       x := 1
       7     end.
```

The listing produced for program 27 with semantics enabled

```
       1     program p027(input, output);
       2       const maxrelations = 2;
       3       var x: integer;
       4           prtlrdrdata: array [1..2*maxrelations] of integer;
                                          --
***   31:   e - replaced '*' with ','
***   32:   e - a constant appears where a type name was expected
       5     begin
       6       x := 1
       7     end.
```

The listing produced for program 27 with semantics disabled

```
      1     program p031(input, output);
      2       var x, loc: integer;
      3       function getelement(x: integer): boolean;
      4       begin
      5         getelement := true;
      6       end;
      7     begin
      8       if no
                  ^
***     6:  e - deleted 'no'
      9       if not getelement(loc) then x := 1
              ^
***     3:  e - deleted 'if'
     10     end.
```

The listing produced for program 31 with semantics enabled

```
      1     program p031(input, output);
      2       var x, loc: integer;
      3       function getelement(x: integer): boolean;
      4       begin
      5         getelement := true;
      6       end;
      7     begin
      8       if no
                  ^
***     6:  e - "no" is undefined
      9       if not getelement(loc) then x := 1
             ^
***     2:  e - inserted 'then'
     10     end.
```

The listing produced for program 31 with semantics disabled

```
       1     program p033(input, output);
       2       var x, sc, numeles: integer;
       3     begin
       4       sc := numeles+1];
                              ^
***   18:    e - deleted ']'
       5       x := 1
       6     end.
```

The listing produced for program 33 with semantics enabled

```
       1     program p033(input, output);
       2       var x, sc, numeles: integer;
       3     begin
       4       sc := numeles+1];
                            ^
***   16:    e - inserted '[' before '1'
***   16:    e - the operands of '+' are not compatible
       5       x := 1
       6     end.
```

The listing produced for program 33 with semantics disabled

```
        1   program p035(input, output);
        2     const listsize = 10;
        3     var x : integer;
        4     procedure intlkdlst(size: integer);
        5     begin
        6     end;
        7   begin
        8     intlkdlst[listsize];
                       ^        ^
***    12:  e - replaced '[' with '('
***    21:  e - replaced ']' with ')'
        9     x := 1
       10   end.
```

The listing produced for program 35 with semantics enabled

```
        1   program p035(input, output);
        2     const listsize = 10;
        3     var x : integer;
        4     procedure intlkdlst(size: integer);
        5     begin
        6     end;
        7   begin
        8     intlkdlst[listsize];
              ^                  ^
***     3:  e - replaced procedure with variable
***    22:  e - deleted ';'
        9     x := 1
              ^
***     3:  e - deleted 'x'
       10   end.
```

The listing produced for program 35 with semantics disabled

```
        1     program p043(input, output);
        2       var max, norel: integer;
        3           x: array [1..2, 1..2] of integer;
        4     begin
        5       begin
        6         if max < x[norel, 2] then
        7           max := x[norel, 2]
        8       end
        9       real;
                ^
***      3:   e - deleted 'real'
       10       max := 1
       11     end.
```

The listing produced for program 43 with semantics enabled

```
        1     program p043(input, output);
        2       var max, norel: integer;
        3           x: array [1..2, 1..2] of integer;
        4     begin
        5       begin
        6         if max < x[norel, 2] then
        7           max := x[norel, 2]
        8       end
                   ^
***      6:   e - inserted ';'
        9       real;
                ^
***      3:   e - a type name appears where a procedure was expected
       10       max := 1
       11     end.
```

The listing produced for program 43 with semantics disabled

```
     1      program p059(input, output);
     2        var x, data: integer;
     3      begin
     4        writeln('-***error*** at least one loop exists in the',
     5                da
                      ^
***   11:   e - deleted 'da'
     6                data);
     7        x := 1
     8      end.
```

The listing produced for program 59 with semantics enabled

```
     1      program p059(input, output);
     2        var x, data: integer;
     3      begin
     4        writeln('-***error*** at least one loop exists in the',
     5                da
                      ^ ^
***   11:   e - "da" is undefined
***   13:   e - inserted ','
     6                data);
     7        x := 1
     8      end.
```

The listing produced for program 59 with semantics disabled

```
     1     program p074(input, output);
             var prime, check, x: integer;
     3     begin
     4       if prime check then x := 1
                     ^
***  11:   e - inserted '='
     5     end.
```

The listing produced for program 74 with semantics enabled

```
     1     program p074(input, output);
     2       var prime, check, x: integer;
     3     begin
     4       if prime check then x := 1
                <----^---->
***   6:   e - boolean expression expected
***  11:   e - inserted '+'
     5     end.
```

The listing produced for program 74 with semantics disabled

```
      1     program p077(input, output);
      2       const a[1] 10; a[2] 15; a[3] 25; a[4] 3;? a[5] 50; a[6] 75;
              <------<->-^---^<->-^---^<->-^----^--^-------------------
***     3:  e - Malformed Declaration
***    10:  e - expression replaced by a constant
***    10:  e - inserted '=' before '['
***    14:  e - deleted '10'
***    18:  e - a is redeclared
***    19:  e - expression replaced by a constant
***    19:  e - inserted '=' before '['
***    23:  e - deleted '15'
***    27:  e - a is redeclared
***    28:  e - expression replaced by a constant
***    28:  e - inserted '=' before '['
***    32:  e - deleted '25'
***    37:  e - inserted '=' before '['
***    40:  e - inserted '+'
      3     begin
      4       x := 1
              ^
***     3:  e - "x" is undefined
      5     end.
```

The listing produced for program 77 with semantics enabled

```
     1     program p077(input, output);
     2       const a[1] 10; a[2] 15; a[3] 25; a[4] 3;? a[5] 50; a[6] 75;
                    <-^^->  ^<-^^->  ^<-^^->  ^<-^^> ^ ^    ^   ^
***   10:  e - expression replaced by a constant
***   10:  e - inserted '=' before '['
***   12:  e - the operands of '+' are not compatible
***   13:  e - inserted '+'
***   18:  e - a is redeclared
***   19:  e - expression replaced by a constant
***   19:  e - inserted '=' before '['
***   21:  e - the operands of '+' are not compatible
***   22:  e - inserted '+'
***   27:  e - a is redeclared
***   28:  e - expression replaced by a constant
***   28:  e - inserted '=' before '['
***   30:  e - the operands of '+' are not compatible
***   31:  e - inserted '+'
***   36:  e - a is redeclared
***   37:  e - expression replaced by a constant
***   37:  e - inserted '=' before '['
***   39:  e - the operands of '+' are not compatible
***   40:  e - inserted '+'
***   43:  e - replaced '?' with 'begin'
***   45:  e - replaced constant with variable
***   49:  e - inserted ':='
***   54:  e - replaced constant with variable
     3     begin
     4       x := 1
             ^
***    3:  e - "x" is undefined
     5     end.
              ^
***    3:  e - inserted 'end' before '.'
```

The listing produced for program 77 with semantics disabled

```
        1     program p078(input, output);
        2       var prime, check, x: integer;
        3     begin
        4       if prime/check trunc(prime/check) then x := 1
                              ^
***    17:    e - inserted '='
        5     end.
```

The listing produced for program 78 with semantics enabled

```
        1     program p078(input, output);
        2       var prime, check, x: integer;
        3     begin
        4       if prime/check trunc(prime/check) then x := 1
                   <----------^------------------>
***     6:    e - boolean expression expected
***    17:    e - inserted '+'
        5     end.
```

The listing produced for program 78 with semantics enabled.

```
      1    program p082(input, output);
      2      var x: integer;
      3      function xfact(x: integer): integer;
      4      begin
      5        xfact := if x = 0 then 1.0 else x * xfact(x - 1)
               ^     ^                ^          ^
***    5: e - a function appears where a procedure was expected
***   11: e - replaced ':=' with ';'
***   28: e - deleted '1.0'
***   39: e - replaced '*' with ':='
      6      end;
      7    begin
      8      x := 1
      9    end.
```

The listing produced for program 82 with semantics enabled

```
      1    program p082(input, output);
      2      var x: integer;
      3      function xfact(x: integer): integer;
      4      begin
      5        xfact := if x = 0 then 1.0 else x * xfact(x - 1)
               ^     ^                ^        ^ ^ ^
***    5: e - a function appears where a procedure was expected
***   11: e - replaced ':=' with ';'
***   28: e - deleted '1.0'
***   37: e - a parameter appears where a procedure was expected
***   39: e - replaced '*' with ';'
***   41: e - a functions appears where a procedure was expected
      6      end;
      7    begin
      8      x := 1
      9    end.
```

The listing produced for program 82 with semantics disabled

```
        1     program p087(input, output);
        2       function f(x: integer): integer;
        3       begin
        4         f ;= if x = 0 then 1 else x * f(x-1);
                  ^  ^               ^^        ^
***     5:    e - a function appears where a procedure was expected
***     8:    e - replaced '=' with ';'
***    23:    e - inserted 'goto'
***    24:    e - label 1 is undeclared
***    33:    e - replaced '*' with ':='
        5       end;
        6     begin
        7     end.
```

The listing produced for program 87 with semantics enabled

```
        1     program p087(input, output);
        2       function f(x: integer): integer;
        3       begin
        4         f ;= if x = 0 then 1 else x * f(x-1);
                  ^  ^               ^^      ^ ^ ^
***     5:    e - a function appears where a procedure was expected
***     8:    e - replaced '=' with ';'
***    24:    e - label 1 is undeclared
***    25:    e - inserted ':'
***    31:    e - a parameter appears where a procedure was expected
***    33:    e - replaced '*' with ';'
***    35:    e - a functions appears where a procedure was expected
        5       end;
        6     begin
        7     end.
```

The listing produced for program 87 with semantics disabled

```
      1    program p091(input, output);
      2      const a[1] = 10; a[2] = 15; a[3] = 25; a[4] == 35; a[5] = 50;
                   ^<-------^-^----^---^------^---^---------->  ^<---^-->
***     9: e - a is redeclared
***    10: e - Missing/Malformed Expression
***    10: e - inserted '=' before '['
***    18: e - replaced ';' with '+'
***    20: e - "a" is undefined
***    25: e - replaced '=' with '+'
***    29: e - replaced ';' with '+'
***    36: e - replaced '=' with '+'
***    40: e - replaced ';' with '+'
***    54: e - a is redeclared
***    55: e - expression replaced by a constant
***    55: e - inserted '=' before '['
***    59: e - the operands of '=' are not compatible
      3            a[6] = 75;
                   ^<---^-->
***     9: e - a is redeclared
***    10: e - expression replaced by a constant
***    10: e - inserted '=' before '['
***    14: e - the operands of '=' are not compatible
      4      var x: integer;
      5    begin
      6      x := 1
      7    end.
```

The listing produced for program 91 with semantics enabled

```
         1      program p091(input, output);
         2        const a[1] = 10; a[2] = 15; a[3] = 25; a[4] == 35; a[5] = 50;
                        <---^-->  ^<---^-->  ^<---^-->  ^<---^^-->  ^<---^-->
    ***   10:   e - expression replaced by a constant
    ***   10:   e - inserted '=' before '['
    ***   14:   e - the operands of '=' are not compatible
    ***   20:   e - a is redeclared
    ***   21:   e - expression replaced by a constant
    ***   21:   e - inserted '=' before '['
    ***   25:   e - the operands of '=' are not compatible
    ***   31:   e - a is redeclared
    ***   32:   e - expression replaced by a constant
    ***   32:   e - inserted '=' before '['
    ***   36:   e - the operands of '=' are not compatible
    ***   42:   e - a is redeclared
    ***   43:   e - expression replaced by a constant
    ***   43:   e - inserted '=' before '['
    ***   47:   e - the operands of '=' are not compatible
    ***   48:   e - deleted '='
    ***   54:   e - a is redeclared
    ***   55:   e - expression replaced by a constant
    ***   55:   e - inserted '=' before '['
    ***   59:   e - the operands of '=' are not compatible
         3              a[6] = 75;
                        ^<---^-->
    ***    9:   e - a is redeclared
    ***   10:   e - expression replaced by a constant
    ***   10:   e - inserted '=' before '['
    ***   14:   e - the operands of '=' are not compatible
         4        var x: integer;
         5      begin
         6        x := 1
         7      end.
```

The listing produced for program 91 with semantics disabled

```
       1     program p096(input, output);
       2       var m: integer;
       3           fact, stirl, x: real;
       4       function pfact(f, x: real; m: integer): real;
       5       begin
       6         pfact := 1.0
       7       end;
       8       function pstirl(f, x: real; m: integer): real;
       9       begin
      10         pstirl := 1.0
      11       end;
      12     begin
      13       begin
      14         write(x, m, pfact(fact, x, m), pstirl(stirl, x, m),
      15               pfact - pstirl)
                       <--->
***     11:  e - a function identifier cannot be an operand of '-'
      16     end.
             ^
***      1:  e - inserted 'end'
```

The listing produced for program 96 with semantics enabled

```
       1     program p096(input, output);
       2       var m: integer;
       3           fact, stirl, x: real;
       4       function pfact(f, x: real; m: integer): real;
       5       begin
       6         pfact := 1.0
       7       end;
       8       function pstirl(f, x: real; m: integer): real;
       9       begin
      10         pstirl := 1.0
      11       end;
      12     begin
      13       begin
      14         write(x, m, pfact(fact, x, m), pstirl(stirl, x, m),
      15               pfact - pstirl)
                       <--->
***     11:  e - a function identifier cannot be an operand of '-'
      16     end.
               ^
***      3:  e - inserted 'end' before '.'
```

The listing produced for program 96 with semantics disabled

```
         1     program p101(input, output);
         2       var prcount, x: integer;
         3     begin
         4       99  prcount := prcount;
                 ^
***       3:   e - deleted '99'
         5       x := 1
         6     end.
```

The listing produced for program 101 with semantics enabled

```
         1     program p101(input, output);
         2       var prcount, x: integer;
         3     begin
         4       99  prcount := prcount;
                 ^ ^
***       3:   e - label 99 is undeclared
***       5:   e - inserted ':'
         5       x := 1
         6     end.
```

The listing produced for program 101 with semantics disabled

```
        1     program p104(input, output);
        2       var x: integer;
        3     begin
        4       begin
        5         x := 1
        6       end;
        7       procedure stirling;
                ^         ^
***      3:   e - deleted 'procedure'
***     13:   e - deleted 'stirling'
        8       begin
        9         x := 1
       10       end;
       11       x := 1
       12     end.
```

The listing produced for program 104 with semantics enabled

```
        1     program p104(input, output);
        2       var x: integer;
        3     begin
        4       begin
        5         x := 1
        6       end;
        7       procedure stirling;
                ^         ^
***      3:   e - deleted 'procedure'
***     13:   e - "stirling" is undefined
        8       begin
        9         x := 1
       10       end;
       11       x := 1
       12     end.
```

The listing produced for program 104 with semantics disabled

```
       1     program p106(input, output);
       2       const n = 10;
       3       var next, kount, x: integer;
       4           arr: array [1..n] of integer;
       5     begin
       6       arr := [2..n];
               <----------->
 ***     3:  e - incompatible assignment
       7       ko nt := 0;
               ^  ^
 ***     3:  e - "ko" is undefined
 ***     5:  e - inserted '.'
       8       next := 2;
       9       x := 1
      10     end.
```

The listing produced for program 106 with semantics enabled

```
       1     program p106(input, output);
       2       const n = 10;
       3       var next, kount, x: integer;
       4           arr: array [1..n] of integer;
       5     begin
       6       arr := [2..n];
               <----------->
 ***     3:  e - incompatible assignment
       7       ko nt := 0;
               ^  ^^
 ***     3:  e - "ko" is undefined
 ***     5:  e - inserted ';'
 ***     6:  e - "nt" is undefined
       8       next := 2;
       9       x := 1
      10     end.
```

The listing produced for program 106 with semantics disabled

```
     1     program p109(input, output);
     2       var i, x: integer;
     3           list: array [1..10] of integer;
     4     begin
     5       readln( list_i?);
                         ^ ^
***   15:  e - replaced '_' with '['
***   17:  e - replaced '?' with ']'
     6       x := 1
     7     end.
```

The listing produced for program 106 with semantics enabled

```
     1     program p109(input, output);
     2       var i, x: integer;
     3           list: array [1..10] of integer;
     4     begin
     5       readln( list_i?);
                     ^   ^ ^
***   11:  e - a variable appears where a function was expected
***   15:  e - replaced '_' with '('
***   17:  e - replaced '?' with ')'
     6       x := 1
     7     end.
```

The listing produced for program 106 with semantics disabled

```
     1     program p112(input, output);
     2       var letter: char;
     3           x: integer;
     4     begin
     5       read(letter);
     6       if letter<>'.' and letter<>' ' then
                      ^ <->           ^ ^
***   12:  e - the operands of '<>' are not compatible
***   14:  e - a character cannot be an operand of 'and'
***   28:  e - deleted '<>'
***   30:  e - deleted '' ''
     7           x := 1
     8     end.
```

The listing produced for program 112 with semantics enabled

```
     1     program p112(input, output);
     2       var letter: char;
     3           x: integer;
     4     begin
     5       read(letter);
     6       if letter<>'.' and letter<>' ' then
                      ^ <->---------->^ <->
***   12:  e - the operands of '<>' are not compatible
***   14:  e - a character cannot be an operand of 'and'
***   14:  e - a boolean value cannot be an operand of '+'
***   28:  e - replaced '<>' with '+'
***   30:  e - a character cannot be an operand of '+'
     7           x := 1
     8     end.
```

The listing produced for program 112 with semantics disabled

```
       1      program p115(input, output);
       2        type alfa = packed array [1..10] of char;
       3        var buf: array [1..10] of char;
       4            a: alfa;
       5            list: array [1..10] of alfa;
       6            t, x: integer;
       7      begin
       8        pack(buf, 1, a);
       9        list(t) := a;
                    ^ ^
***      7:   e - replaced '(' with '['
***      9:   e - replaced ')' with ']'
      10          x := 1
      11      end.
```

The listing produced for program 115 with semantics enabled

```
       1      program p115(input, output);
       2        type alfa = packed array [1..10] of char;
       3        var buf: array [1..10] of char;
       4            a: alfa;
       5            list: array [1..10] of alfa;
       6            t, x: integer;
       7      begin
       8        pack(buf, 1, a);
       9        list(t) := a;
                ^       ^  ^
***      3:   e - a variable appears where a procedure was expected
***     11:   e - replaced ':=' with ';'
***     14:   e - a variable appears where a procedure was expected
      10          x := 1
      11      end.
```

The listing produced for program 115 with semantics disabled

```
      1     program p118(input, output);
      2       var x: integer;
      3     begin;
      4       procedure factr(n: integer; var factor: integer);
              <------------------------> <------------------->
***     3:  e - Malformed Statement
***    31:  e - Malformed Statement
      5       begin
      6         x := 1
      7       end;
      8       x := 1
      9     end.
```

The listing produced for program 118 with semantics enabled

```
      1     program p118(input, output);
      2       var x: integer;
      3     begin;
      4       procedure factr(n: integer; var factor: integer);
              ^         <-----^---------> <------------------->
***     3:  e - deleted 'procedure'
***    13:  e - Malformed Statement
***    13:  e - "factr" is undefined
***    19:  e - "n" is undefined
***    31:  e - Malformed Statement
      5       begin
      6         x := 1
      7       end;
      8       x := 1
      9     end.
```

The listing produced for program 118 with semantics disabled

```
      1     program p123(input, output);
      2       const word = 'hello';
      3       var x, h, cntr, l: integer;
      4           hi: array [1..10] of packed array [1..5] of char;
      5     begin
      6       hi(h) := word;
                ^  ^
***    5:   e - replaced '(' with '['
***    7:   e - replaced ')' with ']'
      7       if h <= 1 then
      8        begin
      9          writeln(?error sort?, cntr, h, l,);
                 <--------------------------------->
***    6:   e - Malformed Statement
     10          goto 1; * abnrm *
                         <-------
***   14:   e - Malformed Statement
     11        end;
     12       x := 1
     13     end.
```

The listing produced for program 123 with semantics enabled

```
      1     program p123(input, output);
      2       const word = 'hello';
      3       var x, h, cntr, l: integer;
      4           hi: array [1..10] of packed array [1..5] of char;
      5     begin
      6       hi(h) := word;
              ^     ^  ^
***    3:   e - a variable appears where a procedure was expected
***    9:   e - replaced ':=' with ';'
***   12:   e - a constant appears where a procedure was expected
      7       if h <= 1 then
      8        begin
      9          writeln(?error sort?, cntr, h, l,);
                 <--------------------------------->
***    6:   e - Malformed Statement
     10          goto 1; * abnrm *
                         <-------
***   14:   e - Malformed Statement
     11        end;
     12       x := 1
     13     end.
```

The listing produced for program 123 with semantics disabled

```
     1      program p125(input, output);
     2        type alfa = packed array [1-10] of char;
                                          ^
***   30:   e - replaced '-' with '..'
     3        var x: integer;
     4      begin
     5        x := 1
     6      end.
```

The listing produce for program 125 with semantics enabled

```
     1      program p125(input, output);
     2        type alfa = packed array [1-10] of char;
                                         <^->
***   29:   e - lower bound exceeds upper bound
***   30:   e - inserted '..' before '-'
     3        var x: integer;
     4      begin
     5        x := 1
     6      end.
```

The listing produced for program 125 with semantics disabled

```
     1      program p126(input, output);
     2        matrixknown(name: char, lower: boolean,
             ^                       ^               ^

***    2:   e - inserted 'procedure'
***   25:   e - replaced ',' with ';'
***   41:   e - replaced ',' with ';'
     3                    var pointer: integer): boolean;
                                             <------->

***   36:   e - return type specified for a procedure
     4          var x: integer;
     5        begin
     6          x := 1
     7        end.
                 ^

***    6:   E - Unrecoverable Syntax Error
```

The listing produced for program 126 with semantics enabled

```
     1      program p126(input, output);
     2        matrixknown(name: char, lower: boolean,
             ^^           ^     ^     ^      ^

***    2:   e - inserted 'begin'
***    3:   e - "matrixknown" is undefined
***   15:   e - "name" is undefined
***   21:   e - type name appears where an expression was expected
***   27:   e - "lower" is undefined
***   34:   e - type name appears where an expression was expected
     3                    var pointer: integer): boolean;
                          ^   ^        ^       ^

***   15:   e - deleted 'var'
***   19:   e - "pointer" is undefined
***   28:   e - type name appears where an expression was expected
***   36:   e - replaced ':' with ';'
     4          var x: integer;
                <------------->

***    5:   e - Malformed Statement
     5        begin
     6          x := 1
                ^

***    5:   e - "x" is undefined
     7        end.
                 ^

***    5:   e - inserted 'end' before '.'
```

The listing produced for program 126 with semantics disabled

Appendix C

Programs for which Berkeley Pascal or the Burke-Fisher System Outperform the Pascal Auditor

This appendix contains listings of the programs in the Ripley-Druseikis suite for which Berkeley Pascal [GHJ79] or the Burke-Fisher system [BF82] produce better recoveries than the Pascal auditor. Some listings have been altered slightly to make them fit within the page margins. Lines that were too long to fit within the margins were split into two lines. The listings for each program are accompanied by an note explaining why the Pascal auditor produced an inferior recovery.

```
      1  program p039(input, output);
      2    var x, m, tim1: integer;
      3  begin
      4    readln(x, m);
      5    tim1:"x;
e ------------^--- Replaced illegal character with a '='
      6    x := 1
      7  end.
```

Berkeley Pascal's listing for program 39

```
      1      program p039(input, output);
      2        var x, m, tim1: integer;
      3      begin
      4        readln(x, m);
      5        tim1:"x;
               <----->
***      3:  e - Malformed Statement
      6        x := 1
      7      end.
```

The Pascal auditor's listing for program 39

Berkeley Pascal treats the compound symbol ':=' as the sequence consisting of the symbol ':' followed by the symbol '='. Hence, it is able to repair the error in this example by replacing the symbol '"' with the symbol '='. The Pascal auditor treats the compound symbol ':=' as a single symbol. Thus, for the Pascal auditor to achieve the effect of the repair chosen by Berkeley Pascal, the Pascal auditor would have to replace the individual symbols ':' and '"' by the symbol ':='. However, the local recovery algorithm used by the Pascal auditor does not possess the ability to replace a sequence of tokens with another token. Therefore, the Pascal auditor is unable to find a viable repair for this error.

```
     1  program p054(input, output);
     2    const listsize = 5;
     3    var listdata: array [1..listsize];
E ----------------------------------------^--- Expected keyword of
E ----------------------------------------^--- Inserted identifier
     4          x: integer;
     5  begin
     6    x := 1
     7  end.
```

Berkeley Pascal's listing for program 54

```
    1   program p054(input, output);
    2     const listsize = 5;
    3     var listdata: array [1..listsize];
                                          ^
*** Syntax Error: Unexpected input --
                  "OF IDENTIFIER" inserted to match "ARRAY" on line 3
    4        x: integer;
    5   begin
    6     x := 1
    7   end.
```

The Burke-Fisher system's listing for program 54

```
     1      program p054(input, output);
     2        const listsize = 5;
     3        var listdata: array [1..listsize];
                            <------------------>
***    17:  e - Missing/Malformed Type
     4            x: integer;
     5      begin
     6        x := 1
     7      end.
```

The Pascal auditor's listing for program 54

Both Berkeley Pascal and the Burke-Fisher system are able to insert two tokens in some contexts. The Pascal auditor's local recovery algorithm is unable to insert more than one token under any circumstances. This example demonstrates that the ability to do multiple insertions sometimes leads to better recoveries. Because the only viable repair for the error in this example consists of inserting two tokens, the Pascal auditor is forced to resort to panic mode.

```
     1  program p055(input, output);
     2    const listsize = 5;
     3    var listptr := array [1..listsize];
e --------------------^--- Deleted '='
E ------------------------------------------^--- Expected keyword of
E ------------------------------------------^--- Inserted identifier
     4        x: integer;
     5  begin
     6    x := 1
     7  end.
```

Berkeley Pascal's listing for program 55

```
    1   program p055(input, output);
    2     const listsize = 5;
    3     var listptr := array [1..listsize];
                      ^                    ^

*** Syntax Error: ":" expected instead of ":="
*** Syntax Error: Unexpected input --
                 "OF IDENTIFIER" inserted to match "ARRAY" on line 3
    4         x: integer;
    5   begin
    6     x := 1
    7   end.
```

The Burke-Fisher system's listing for program 55

```
     1      program p055(input, output);
     2        const listsize = 5;
     3        var listptr := array [1..listsize];
                          ^  <----------------->
***    15:  e - replaced ':=' with ':'
***    18:  e - Missing/Malformed Type
     4            x: integer;
     5      begin
     6        x := 1
     7      end.
```

The Pascal auditor's listing for program 55

This example simply repeats the lesson of the previous example.

```
     1  program p069(input, output);
     2    var sub, x, f: integer;
     3    count, listdata: array [1..10] of integer;
     4  begin
     5    if count[listdata[sub] := 0 then
e -------------------------------^--- Replaced ':' with a ']'
     6      begin
     7        f := listdata[sub];
     8      end;
     9    x := 1
    10  end.
```

Berkeley Pascal's listing for program 69

```
    1   program p069(input, output);
    2     var sub, x, f: integer;
    3     count, listdata: array [1..10] of integer;
    4   begin
    5     if count[listdata[sub] := 0 then
                                 ^ ^
*** Syntax Error: "=" expected instead of ":="
*** Syntax Error: "]" expected after this token
    6       begin
    7         f := listdata[sub];
    8       end;
    9     x := 1
   10   end.
```

The Burke-Fisher system's listing for program 69

```
     1     program p069(input, output);
     2       var sub, x, f: integer;
     3       count, listdata: array [1..10] of integer;
     4     begin
     5       if count[listdata[sub] := 0 then
               <---------------------->
***     6:  e - Missing/Malformed Expression
     6         begin
     7           f := listdata[sub];
     8         end;
     9       x := 1
    10     end.
```

The Pascal auditor's listing for program 69

Berkeley Pascal and the Burke-Fisher both handle the error in program 69 better than the Pascal auditor. However, they find different repairs for the error. Berkeley Pascal is able to repair the error by replacing the colon ':' in the symbol ':=' by a right bracket ']'. This repair is possible because Berkeley Pascal treats the compound symbol ':=' as two separate symbols (see the note for program 39 earlier in this appendix). Neither the Burke-Fisher system nor the Pascal auditor are able to repair the error in this manner because they treat the symbol ':=' as a single token.

The Burke-Fisher system repairs the error by replacing the symbol ':=' with the symbol '=' and then inserting a right bracket following the constant 0. These are really two separate repairs. The Burke-Fisher system will invoke its error recovery algorithm twice for this example: once when it read the symbol ':=', and again when it reads the keyword **then**. The Pascal auditor is unable to repair the error in this way because it requires that the parser must be able to shift at least two tokens following a repair for that repair to be considered viable. Therefore, it is unable to replace the symbol ':=' by the symbol '=', because the parser is unable to shift the keyword **then** following that repair.

```
          1  program p073(input, output);
          2    var check, prime, x: integer;
          3  begin
          4    check: 1?
     E ---------------^--- Expected '='
          5    begin
          6      while check)' prime do x := 1
     E ----------------------^--- Unmatched ' for string
          4    check: 1?
     e ----------------^--- Replaced illegal character with a ';'
          6      while check)' prime do x := 1
     E ---------------------^--- Missing/malformed expression
          7    end
          8  end.
```

Berkeley Pascal's listing for program 73

```
          1     program p073(input, output);
          2       var check, prime, x: integer;
          3     begin
          4       check: 1?
                  <-------
     ***    3:  e - Malformed Statement
          5       begin
          6         while check)' prime do x := 1
                          <----------->
     ***   11:  e - Missing/Malformed Expression
          7       end
          8     end.
```

The Pascal auditor's listing for program 73

Berkeley Pascal finds better repairs for the errors on line 4 than does the Pascal auditor. Although the listing that Berkeley Pascal generates does not made it clear, Berkeley Pascal repairs the errors by inserting the character '=' following the character ':' and by replacing the character '?' with the token ';'. Yet again, the fact that Berkeley Pascal treats the compound symbol ':=' as two separate symbols allows the repair to be done. The Pascal auditor could not have produced the same repair even if it treated the symbol ':=' as separate symbols because it would not be able to parse far enough after the repair for the repair to be considered viable.

```
    1  program p093(input, output);
    2    var i, x: integer;
    3  begin
    4    repeat
    5      x := 1
    6    until -[sqrt(i);
              ^
*** Syntax Error: Unexpected "[" ignored
    7      x := 1
    8  end.
```

The Burke-Fisher system's listing for program 93

```
      1     program p093(input, output);
      2       var i, x: integer;
      3     begin
      4       repeat
      5         x := 1
      6       until -[sqrt(i);
                    <<<----->^
***     9:  e - boolean expression expected
***    10:  e - numeric expression expected
***    11:  e - set member type is not ordinal
***    18:  e - inserted ']' before ';'
      7       x := 1
      8     end.
```

The Pascal auditor's listing for program 93

Unlike either the Pascal auditor or Berkeley Pascal, the Burke-Fisher system is able to repair errors by deleting or replacing tokens that the parser has shifted but has not yet used in a reduction. In this example, the Burke-Fisher system is able to find a better repair for the syntax error detected on line 6 because it is able to delete the left bracket '['. The Pascal auditor is unable to make the same repair because the error is not detected until the semicolon ';' has been read. Therefore, the Pascal auditor will have already shifted all symbols up but not including the identifier *i* by the time it detects the error.

```
        1   program p097(input, output);
        2     label 2;
        3     var count, x: integer;
        4   begin
        5     begin
        6       count := 0;
        7       go to 2
                ^
    *** Syntax Error: "GOTO" expected
        8     end;
        9   end.
```

The Burke-Fisher system's listing for program 97

```
          1     program p097(input, output);
          2       label 2;
          3       var count, x: integer;
          4     begin
          5       begin
          6         count := 0;
          7         go to 2
                    ^  ^
    ***     5:  e - "go" is undefined
    ***     8:  e - replaced 'to' with ':='
          8       end;
          9     end.
```

The Pascal auditor's listing for program 97

The Burke-Fisher error recovery algorithm considers some types of repairs that are not considered by the Pascal auditor. In particular, it is able to merge the texts of adjacent tokens to form a new token. Burke and Fisher call this form of repair *token merging*. In this example, the Burke-Fisher system is able to merge the identifier *go* and the keyword **to** to form the keyword **goto**. The Pascal auditor does not do token merging. Therefore, it is unable to repair the error in this example.

```
     1  program p101(input, output);
     2    var prcount, x: integer;
     3  begin
     4    99  prcount := prcount;
e ------------^--- Inserted ':'
     5    x := 1
     6  end.
E 4 - 99 is undefined
```

Berkeley Pascal's listing for program 101

```
    1   program p101(input, output);
    2     var prcount, x: integer;
    3   begin
    4     99  prcount := prcount;
          ^
*** Syntax Error: ":" expected after this token
    5     x := 1
    6   end.
```

The Burke-Fisher system's listing for program 101

```
     1      program p101(input, output);
     2        var prcount, x: integer;
     3      begin
     4        99  prcount := prcount;
              ^
***     3:  e - deleted '99'
     5        x := 1
     6      end.
```

The Pascal auditor's listing for program 101

Program 101 demonstrates that using semantics to guide the choice of a repair can result in inferior recoveries. The number 99 on line 4 was probably intended to be a label. However, because 99 has not been declared to be a label and because it has not been used in a goto-statement, the Pascal auditor does not consider it a label. Therefore, when the Pascal auditor tries inserting a colon ':' after the number 99, it detects a semantic error. Since deleting the number 99 results in a program that is both semantically and syntactically correct, the Pascal auditor chooses that repair over inserting a colon.

```
     1  program p123(input, output);
     2    const word = 'hello';
     3    var x, h, cntr, l: integer;
     4        hi: array [1..10] of packed array [1..5] of char;
     5  begin
     6    hi(h) := word;
E --------^--- Replaced variable id with a procedure id
E --------------^--- Malformed statement
     7    if h <= 1 then
     8     begin
     9       writeln(?error sort?, cntr, h, l,);
E -------------------^--- Illegal character
E ------------------------------^--- Illegal character
E -------------------------------------------^--- Deleted ','
    10       goto 1; * abnrm *
E -------------------^--- Malformed statement
    11    end;
    12    x := 1
    13  end.
E 10 - 1 is undefined
```

Berkeley Pascal's listing for program 123

```
     1     program p123(input, output);
     2       const word = 'hello';
     3       var x, h, cntr, l: integer;
     4           hi: array [1..10] of packed array [1..5] of char;
     5     begin
     6       hi(h) := word;
               ^ ^
***    5:  e - replaced '(' with '['
***    7:  e - replaced ')' with ']'
     7       if h <= 1 then
     8        begin
     9          writeln(?error sort?, cntr, h, l,);
                <-------------------------------->
***    6:  e - Malformed Statement
    10          goto 1; * abnrm *
                        <-------
***   14:  e - Malformed Statement
    11       end;
    12       x := 1
    13     end.
```

The Pascal auditor's listing for program 123

Berkeley Pascal is unusual in that its lexical analyzer recognizes the character '?' as a string quote. Whenever '?' appears, the lexical analyzer issues an error message announcing that it has found an illegal character. It then checks if there is another occurrence of the character '?' on the same line, and if so, constructs a string from the text delimited by the two question marks. Thus, Berkeley Pascal is easily able to handle the errors on line 9. The Pascal auditor, on the other hand, treats the character '?' just as it would any other illegal character, and so it has to resort to panic mode. Therefore, it does not detect the later error on the same line.

Berkeley Pascal also does better than the Pascal auditor in that it notes that the label 1 has not been declared. Because the Pascal auditor does LR(2) error checking, it detects the error on line 10 before it has finished reducing the goto statement. The fact that the label is undeclared is detected and an error message is sent to the standard routine for reporting errors. However, because the error is detected while the Pascal auditor is executing it panic mode algorithm, the error message is suppressed.

Appendix D

Some Examples for which the Pascal Auditor Produces Better Recoveries than Berkeley Pascal

This appendix contains the listings generated for a sample of the programs for which the Pascal auditor produces better recoveries than does Berkeley Pascal [GHJ79]. Some listings had to be modified to make all the lines fit within the margins of the page. Also, cautionary warning messages have been deleted. (A cautionary warning message is a warning message caused by a suspicious but legal construct). The listings for each program are accompanied by a note explaining the reasons for the differences in the recoveries.

```
     1  program p005(input, output);
     2    function getelement(var x: integer); boolean;
E 2 - Function type must be specified
e -------------------------------------------^---
          Replaced identifier with a keyword forward
     3      var q: integer;
w 3 - Variable declarations should precede routine declarations
     4    begin
     5      x := 1
E ----------^--- Undefined variable
     6    end;
E -----------^--- Expected '.'
     7  begin
     8  end.
In program p:
  E - Unresolved forward declaration of function getelement
  E - x undefined on line 5
     6    end;
E -----------^--- End-of-file expected - QUIT
```

Berkeley Pascal's listing for program 5

```
     1     program p005(input, output);
     2       function getelement(var x: integer); boolean;
                                                 ^
***   38:  e - replaced ';' with ':'
     3         var q: integer;
     4       begin
     5         x := 1
     6       end;
     7     begin
     8     end.
```

The Pascal auditor's listing for program 5.

This example illustrates the harm caused by default reductions. Unlike ANSI Pascal [ANS83], Berkeley Pascal treats the symbol "forward" as a keyword. The grammar for Berkeley Pascal prohibits any symbol from appearing after the first semicolon on line 2 other than the keyword **forward** and another nonstandard keyword. Therefore, if Berkeley Pascal's parser did not do default reductions, it could not do any reductions involving the first semicolon on line 2 before detecting a syntax error. However, because Berkeley Pascal does do default reductions, it reduces the entire function header including the semicolon before it detects the error. Since the semicolon has been used in a reduction, Berkeley Pascal's local recovery algorithm cannot delete or replace it. Therefore, it must find the best repair it can without changing the text up to and including the semicolon.

If the symbol "forward" had been treated as a keyword in the Pascal auditor, the error in this example would have been easy to handle. Any parser that avoids erroneous default reductions would catch the error in time to permit the semicolon to be replaced by a colon without requiring backtracking. Since the Pascal auditor does LR(2) error checking, it will never perform an erroneous default reduction. However, the Pascal auditor treats the symbol "forward" as an identifier rather than as a keyword. Hence, the error is detected by a semantic guard. Therefore, the reason it is possible to replace the semicolon in this example is that the parser for the Pascal auditor does no reductions involving the semicolon or the identifier "boolean" before the semantic guard is executed.

```
     1  program p011(input, output);
     2    var x, nonprime: integer;
     3        numprime: array [1..10] of integer;
     4  begin
     5    if nonprime = 0 then numprime,x. := numprime(x) + 1;
E ---------------------------------------^--- Malformed statement
     6    x := 1
     7  end.
```

Berkeley Pascal's listing for program 11

```
     1     program p011(input, output);
     2       var x, nonprime: integer;
     3           numprime: array [1..10] of integer;
     4     begin
     5       if nonprime = 0 then numprime,x. := numprime(x) + 1;
                                          ^ ^            ^ ^
***   32:  e - replaced ',' with '['
***   34:  e - replaced '.' with ']'
***   47:  e - replaced '(' with '['
***   49:  e - replaced ')' with ']'
     6       x := 1
     7     end.
```

The Pascal auditor's listing for program 11

Program 11 illustrates the advantages of using bracket repair. When Berkeley Pascal considers repairing the error in this program by replacing the comma ',' by a left bracket '[', it discovers a second error upon reading the dot '.'. The reason Berkeley Pascal discovers the second error at that point is that it distinguishes between scalar variables and record variables syntactically. Thus, the error that Berkeley Pascal detects is that a scalar variable has been followed by a dot. Because of the second error, Berkeley Pascal decides that replacing the comma by a left bracket is not a viable repair.

The Pascal auditor also tries replacing the comma by a left bracket, and it too decides that that repair is not viable. In fact, it fails to find any viable single token repairs. However, when it tries replacing bracket repairs, it finds that replacing the comma by a left bracket and the dot by a right bracket allows parsing to continue for an acceptable distance.

When the Pascal auditor attempts to reduce the second occurrence of the identifier *numprime* on line 5 to a function name, it discovers a semantic error that triggers the syntactic error recovery algorithm. None of the single token repairs attempted are found to be satisfactory, but the bracket repair consisting of replacing the parentheses with the corresponding square brackets is found to be the best repair for the semantic error.

```
    1  program p020(input, output);
    2    funtion getelement(var x: integer): boolean;
e --------^--- Replaced identifier with a keyword procedure
E 2 - Procedures do not have types, only functions do
    3      var q: integer;
    4    begin
    5      x := 1
    6    end;
    7  begin
    8  end.
```

Berkeley Pascal's listing for program 20

```
    1      program p020(input, output);
    2        funtion getelement(var x: integer): boolean;
             ^
***     3:  e - replaced 'funtion' with 'function'
    3          var q: integer;
    4        begin
    5          x := 1
    6        end;
    7      begin
    8      end.
```

The Pascal auditor's listing for program 20

This example shows the value of the Pascal auditor's spelling matcher. Berkeley Pascal attempts replacing the identifier *funtion* with the keyword **function** and with the keyword **procedure.** Both repairs are found to be viable, but because procedures tend to be more common than functions, Berkeley Pascal chooses to replace the identifier *funtion* with the keyword **procedure.** Had Berkeley Pascal used a spelling matcher to help determine the costs of the repairs, as does the Pascal auditor, it would have been able to recognize that, in this instance, it is better to replace the identifier with the keyword **function.**

For this particular program, the spelling matcher is not the sole factor that causes the Pascal auditor to favor the keyword **function** over the keyword **procedure.** Because the Pascal auditor inspects up to 12 tokens during a forward move, it discovers a semantic error when it reaches the return type specifier at the end of the procedure header. The error is detected semantically because there is an error production that allows a return type specifier in a procedure header (see the explanation of rule 133 in Appendix A). However, if there had been more than 12 tokens between the detection point of the first error and the return type specifier, the spelling matcher alone would have caused the Pascal auditor to favor the keyword *function* over the keyword **procedure.**

```
    1  program p023(input, output);
    2    var m, x: integer;
    3        ffact: real;
    4    function fact(n: integer): integer;
    5      begin
    6        fact := 1
    7      end;
    8    function power(k, y: integer): integer;
    9      begin
   10        power := 1
   11      end;
   12  begin
   13    ffact"(power(m, x) * exp(-m)) div (fact(x));
E -------------^--- Illegal character
   14    x := 1
   15  end.
```

Berkeley Pascal's listing for program 23

```
    1      program p023(input, output);
    2        var m, x: integer;
    3            ffact: real;
    4        function fact(n: integer): integer;
    5          begin
    6            fact := 1
    7          end;
    8        function power(k, y: integer): integer;
    9          begin
   10            power := 1
   11          end;
   12      begin
   13        ffact"(power(m, x) * exp(-m)) div (fact(x));
                  ^
***     8:  e - replaced '"' with ':='
   14        x := 1
   15      end.
```

The Pascal auditor's listing for program 23

Three of the examples in Appendix C showed that treating the compound symbol ':=' as two separate tokens could lead to better recoveries. Program 23 shows that treating ':=' that way can also lead to inferior recoveries. The Pascal auditor can easily replace the illegal token '"' with the symbol ':=' because it treats ':=' as a single token. Berkeley Pascal, however, would have to replace the token '"' with the pair ':' and '='. Berkeley Pascal does not attempt any such multi-symbol repairs. The only other repair that might appear to be viable is to delete the token '"'. However, since Berkeley Pascal has reduced the identifier *ffact* to a variable name, that repair immediately leads to detection of a new syntax error. Therefore, Berkeley Pascal is forced to resort to panic mode.

```
    1  program p035(input, output);
    2    const listsize = 10;
    3    var x : integer;
    4    procedure intlkdlst(size: integer);
    5    begin
    6    end;
    7  begin
    8    intlkdlst[listsize];
E --------^--- Replaced procedure id with a array id
E ---------------------------^--- Expected ':'
    9    x := 1
   10  end.
In program p035:
  E - intlkdlst improperly used on line 8
```

Berkeley Pascal's listing for program 35

```
    1      program p035(input, output);
    2        const listsize = 10;
    3        var x : integer;
    4        procedure intlkdlst(size: integer);
    5        begin
    6        end;
    7      begin
    8        intlkdlst[listsize];
                      -        -
***   12:  e - replaced '[' with '('
***   21:  e - replaced ']' with ')'
    9        x := 1
   10      end.
```

The Pascal auditor's listing for program 35

Program 35 at first appears to be another demonstration of the advantage of performing bracket repairs. However, closer examination of the workings of Berkeley Pascal reveals that the poor recovery from the errors in this program are due to badly chosen costs. Berkeley Pascal assigns a cost of 3 to replacing a procedure identifier with an array identifier. The cost assigned to replacing a left square bracket with a left parenthesis is 10. Therefore, even if bracket repair were considered, it would be rejected. This contention is more clearly shown by a slightly modified example. The following listing was produced by Berkeley Pascal:

```
      1  program p035(input, output);
      2    const listsize = 10;
      3    var x : integer;
      4    procedure intlkdlst(size: integer);
      5    begin
      6    end;
      7  begin
      8    intlkdlst[listsize + 1);
E --------^--- Replaced procedure id with a array id
E -----------------------------^--- Missing/malformed expression
      9    x := 1
     10  end.
In program p035:
  E - intlkdlst improperly used on line 8
```

Here, it is clearly best to replace the left square bracket with a left parenthesis. However, because of the poor choice of costs, it still chooses to replace the procedure identifier *intlkdlst* with an array identifier.

```
1  program p074(input, output);
     2    var prime, check, x: integer;
     3  begin
     4    if prime check then x := 1
e -----------------^--- Inserted '+'
     5  end.
E 4 - Type of expression in if statement must be Boolean, not integer
```

Berkeley Pascal's listing for program 74

```
     1      program p074(input, output);
     2        var prime, check, x: integer;
     3      begin
     4        if prime check then x := 1
                      ^
***   11:   e - inserted '='
     5      end.
```

The Pascal auditor's listing for program 74

Program 74 shows the advantage gained through the use of general static semantic information during error recovery. Both the Pascal auditor and Berkeley Pascal assign a lower cost to inserting the symbol '+' than to inserting the symbol '='. However, when the Pascal auditor tries inserting a '+', it discovers that a semantic error will be detected later. Therefore, it increases the cost of inserting the symbol '+'. and so inserting the symbol '=', which does not lead to a semantic error, turns out to be the least cost repair. Since Berkeley Pascal does not check for general static semantic errors until after parsing has been completed, it does not detect the semantic error until it is too late to affect the choice of recoveries.

```
     1  program sort119(input, output);
     2    const limit = 100;
     3    limitp1 = limit + 1;
E -----------------------------^--- Expected ';'
e -----------------------------^--- Replaced '+' with a keyword label
     4    var x: integer;
w 3 - Label declarations should precede const,
        type, var and routine declarations
     5  begin
     6    x := 1
     7  end.
In program sort119:
  E - label 1 was declared but not defined
```

Berkeley Pascal's listing for program 119

```
     1     program sort119(input, output);
     2       const limit = 100;
     3             limitp1 = limit + 1;
                             <------->
***   19:  e - expression replaced by a constant
     4       var x: integer;
     5     begin
     6       x := 1
     7     end.
```

The Pascal auditor's listing for program 119

The differences in the recoveries produced for program 119 are the result of the Pascal auditor's more thorough use of error productions. Berkeley Pascal does not take advantage of error productions to handle common errors that are beyond the capabilities of its local recovery algorithm. The Pascal auditor is more complete in this respect, though even its handling of error productions could be improved (see Section 10.2).

```
     1  program p125(input, output);
     2    type alfa = packed array [1-10] of char;
e ------------------------------------^--- Inserted '..'
E 2 - Range lower bound exceeds upper bound
     3    var x: integer;
     4  begin
     5    x := 1
     6  end.
```

Berkeley Pascal's listing for program 125

```
     1     program p125(input, output);
     2       type alfa = packed array [1-10] of char;
                                         ^
***  30:   e - replaced '-' with '..'
     3       var x: integer;
     4     begin
     5       x := 1
     6     end.
```

The Pascal auditor's listing for program 125

The recovery that the Pascal auditor produces for program 125 is the most complex application of semantic information for any program in the test sample. The Pascal auditor does not choose to insert the symbol '..' before the hyphen because that repair will lead to a later semantic error. It chooses to replace the hyphen instead because that repair does not result in any errors during the forward move. Berkeley Pascal, the Burke-Fisher system, and the Pascal auditor with semantics disabled all choose to repair this error by inserting the symbol '..' before the hyphen. That repair is the natural choice since insertions are normally less costly than replacements.

References

[AJU75] Aho, Alfred V., Johnson, Stephen C., and Ullman, Jeffrey D., "Deterministic Parsing of Ambiguous Grammars," *Communications of the ACM* 18:8, pp. 441-452, 1975

[Amm81] Ammann, U., "The Zurich Implementation," In *Pascal — The Language and Its Implementation*, edited by D. W. Barron, pp. 63-82, John Wiley and Sons, Ltd., Chichester, 1981

[ANS74] *American National Standard Programming Language COBOL*, ANSI X3.23-1974, American National Standards Institute, New York, 1974

[ANS78] *American National Standard Programming Language FORTRAN*, ANSI X3.9-1978, American National Standards Institute, New York, 1978

[ANS83] *American National Standard Pascal Computer Programming Language*, ANSI/IEEE770X3.97-1983, American National Standards Institute, New York, 1983

[AP72] Aho, Alfred V. and Peterson, Thomas G., "A Minimum Distance Error Correcting Parser for Context-free Languages," *SIAM Journal of Computing* 1:4, pp. 305-312, 1972

[AU77] Aho, Alfred V. and Ullman, Jeffery D., *Principles of Compiler Design*, Addison-Wesley, Reading, Mass., 1977

[BM82] Burke, Michael and Fisher, Gerald A., Jr., "A Practical Method for Syntactic Error Diagnosis and Recovery," *SIGPLAN Notices* 17:6, pp. 67-78, 1982

[Bro82] Brown, Peter J., "'My System Gives Excellent Error Messages' — Or Does It?," *Software — Practice and Experience* 12:1, pp. 91-94, 1982

[Bro83] Brown, Peter J., "Error Messages: the Neglected Area of the Man/Machine Interface," *Communications of the ACM* 26:4, pp. 246-249, 1983

[Cul69] Culik, Karel, II, *Attributed Grammars and Languages*, Publication No. 3, Dèpartment d'Informatique, Université de Montreal, May 1969

[DoD83] U. S. Department of Defense, *Military Standard — Ada Programming Language*, ANSI/MIL-STD-1815A, U. S. Government Printing Office, Washington, D. C., 1983

[DP82] DeRemer, Franklin L. and Pennello, Thomas J., "Efficient Computation of LALR(1) Lookahead Sets," *ACM Transactions on Programming Languages and Systems* 4:4, pp. 615-649, 1982

[DR76] Druseikis, Frederick C. and Ripley, G. David, "Error Recovery for Simple LR(k) Parsers," *Proceedings of the Annual Conference of the ACM*, pp. 396-400, ACM, Inc., New York, N. Y., 1976

[FL76] Feyock, Stefan and Lazarus, Paul, "Syntax-directed Correction of Syntax Errors," *Software — Practice and Experience* 6:2, pp. 207-219, 1976

[FM80] Fischer, Charles N. and Mauney, Jon, "On the Role of Error Productions in Syntactic Error Correction," *Computer Languages* 5:3/4, pp. 131-140, 1980

[FMQ80] Fischer, Charles N., Milton, Donn R., and Quiring, Sam B. "Efficient LL(1) Error Correction and Recovery Using Only Insertions," *Acta Informatica* 13:2, pp. 141-154, 1980

[GHJ79] Graham, Susan L., Haley, Charles B., and Joy, William N., "Practical LR Error Recovery," *SIGPLAN Notices* 14:8, pp. 168-175, 1979

[GKM83] Graham, Susan L., Kessler, Peter B., and McKusick, Marshall K., "An Execution Profiler for Modular Programs," *Software — Practice and Experience* 13:8, pp. 671-685, 1983

[GR75] Graham, Susan L. and Rhodes, Steven Paul, "Practical Syntactic Error Recovery," *Communications of the ACM* 18:11, pp. 639-650, 1975

[Har77] Hartmann, A. C., *A Concurrent Pascal Compiler for Minicomputers*, Lecture Notes in Computer Science No. 50, Springer-Verlag, Berlin, 1977

[Joh75] Johnson, Stephen C., "Yacc: Yet Another Compiler-Compiler," *Computer Science Technical Report* 32, Bell Laboratories, Murray Hill, N. J., 1975

[KR78] Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language*, Prentice-Hall, Englewood Cliffs, N. J., 1978

[Knu68] Knuth, Donald E., "Semantics of Context-free Languages," *Mathematical Systems Theory* 2:2, pp. 127-145, 1968, *Corrigenda* in *Mathematical Systems Theory* 5:1, pp. 95-96, 1971

[Kos71] Koster, C. H. A., "Affix Grammars," In *Algol 68 Implementation*, edited by J. E. L. Peck, North-Holland, Amsterdam, 1971

[Kos73] Koster, C. H. A., "Error Reporting, Error Treatment, and Error Correction in ALGOL Translation, Part 1," In *Gesellschaft für Informatik e. V. — 2. Jahrestagung 1972*, edited by P. Deussen, Lecture Notes in Economics and Mathematical Systems No. 78, Springer-Verlag, Berlin, 1973

[Lév75] Lévy, Jean-Pierre, "Automatic Correction of Syntax-Errors in Programming Languages," *Acta Informatica* 4:3, pp. 271-292, 1975

[LRS74] Lewis, P. M., Rosenkrantz, D. J., and Stearns, R. E., "Attributed Translations," *Journal of Computer and Systems Sciences* 9:3, pp. 297-307, 1974

[Lyo74] Lyon, G., "Syntax-directed Least-errors Analysis for Context-free Languages: a Practical Approach," *Communications of the ACM* 17:1, pp. 3-14, 1974

[MHW70] McKeeman, W. M., Horning J. J., and Wortman, D. B., *A Compiler Generator*, Prentice-Hall, Englewood, N. J., 1970

[MKR79] Milton, D. R., Kirchhoff, L. W., and Rowland, B. R., "An ALL(1) Compiler Generator," *SIGPLAN Notices* 14:8, pp. 144-157, 1979

[MM78] Mickunas, M. Dennis and Modry, John A., "Automatic Error Recovery for LR Parsers," *Communications of the ACM* 21:6, pp. 459-465, 1978

[Mor70] Morgan, Howard L. "Spelling Correction in Systems Programs," *Communications of the ACM* 13:2, pp. 90-94, 1970

[PD77] Pennello, Thomas J. and DeRemer, Franklin L., "Practical Error Recovery for LR Parsers," Board of Studies in Information Sciences, University of California at Santa Cruz, Santa Cruz, December 1977

[PD78] Pennello, Thomas J. and DeRemer, Franklin L., "A Forward Move for LR Error Recovery," *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pp. 241-254, 1978

[PK80] Pai, Ajit B. and Kieburtz, Richard B. "Global Context Recovery: a New Strategy for Syntactic Error Recovery by Table-Driven Parsers," *ACM Transactions on Programming Languages and Systems* 2:1, pp. 18-41, 1980

[Poh83] Pohlmann, Werner, "LR Parsing for Affix Grammars," *Acta Informatica* 20:4, pp. 283-300, 1983

[RD78] Ripley, G. David and Druseikis, Frederick C., "A Statistical Analysis of Syntax Errors," *Computer Languages* 3:4, pp. 227-240, 1978

[Sch82] Schmauch, Cosima, *Ein Fehlerbehandlungsalgorithmus für LR-Attributierte Grammatiken*, Fachbereich der Informatik, Universität Kaiserslautern, 1982

[SS83] Sippu, Seppo and Soisalon-Soininen, Eljas, "A Syntax-Error-Handling Technique and Its Experimental Analysis," *ACM Transactions on Programming Languages and Systems* 5:4, pp. 656-679, 1983

[TY79] Tarjan, Robert Endre and Yao, Andrew Chi-Chih, "Storing a Sparse Table," *Communications of the ACM* 22:11, pp. 606-611, 1979

[Wat77] Watt, David Anthony, "The Parsing Problem for Affix Grammars," *Acta Informatica* 8:1, pp. 1-20, 1977

[Wet81] Wetherell, Charles S., "Problems with the Ada Reference Grammar," *SIGPLAN Notices* 16:9, pp. 90-104, 1981

[Wir83] Wirth, Niklaus, *Programming in Modula-2,* 2nd edition, Springer-Verlag, Berlin, 1983