CHAPTER 5.D.

WHAT THE COMPILER SHOULD TELL THE USER

James J. Horning
University of Toronto
Toronto, Canada

0. INTRODUCTION

"When I use a word," Humpty Dumpty said, in a rather scornful tone, "it means just what I choose it to mean - neither more nor less."

"The question is," said Alice, "whether you can make words mean so many different things."

"The question is," said Humpty Dumpty, "which is to be master - that's all."

- Lewis Carroll, Through the Looking Glass.

The emphasis of this course has been on the process of translation from high-level to low-level languages: man-machine communication. It is all too easy to neglect the importance of communication in the other direction, but without effective machine-man communication, compilers are useless. Furthermore, if we are to be truly masters of our machines, we must insist that both parts of this dialogue be conducted in languages acceptable to us.

Koster [1972] identifies several tasks of a compiler:

- to check whether the source program is correct
- if it is not correct, to give all useful information for correcting it (treatment of static errors)
- if it is correct, to translate the program into an equivalent object code program
- to cause the object program to be executed
- to report any errors occurring during execution.

This chapter is concerned with the design of appropriate communication from the compiler to the user regarding each of these tasks. It is motivated by the warning that "the naive programmer is only too willing to confuse the properties of the language with those of the compiler" [Koster 1972].

Since conversations with the compiler are initiated by the user, compiler responses may conveniently be thought of as feedback. It is useful to have both negative feedback (warnings of detected errors), and positive feedback (indications that certain types of errors were not found). Compiler output has both immediate uses (e.g., debugging) and deferred uses (e.g., maintenance, documentation). A well-engineered compiler will take all of these into consideration, without drowning the user in paper.

1. NORMAL OUTPUT

Most of this chapter is devoted to compiler responses to various sorts of errors, but it is worth noting that a basic requirement on any compiler is that it produce suitable output for correct programs. Since error messages and diagnostics are generally in addition to normal output, they constitute a small fraction of the total compiler output, even though errors are detected in most programs.

1.1. HEADINGS

Every compiler should clearly identify itself at the start of each compilation. Failure to provide such identification is one of the most common (and inexcusable) errors of amateur compiler-writers. Useful information includes the

- source language
- target machine
- compiler (name and version)
- date when the compiler was created
- time of the current compilation
- options in effect, and options suppressed.

As illustrated in Figure 1.1., such a heading need not consume much space, and is not hard to generate.

```
TOOLCOM/390 - UNIVERSITY OF TORONTO  VERSION 4.2 (1985 MARCH 22)
SOURCE LANGUAGE: TOOLKIT - TARGET MACHINE: SYSTEM/390 - TIME: 1985 MARCH 29, 14:38
OPTIONS: ON (PARAGRAPH, CROSS-REFERENCE, PROFILE) - OFF (TARGET CODE, TRACE, DUMP)
------------------------------------------------------------------------------------
     1 |   << SOURCE PROGRAM STARTS HERE >>
```

Figure 1.1. A Sample Heading

Headings are perhaps neglected because they don't seem to be too helpful as immediate feedback to the user debugging his program. By and large he treats them as so much noise to be ignored at the top of every listing. However, in the deferred uses of compiler output some piece of identifying information may well be crucial, and extremely hard to reconstruct after the fact.

1.2. LISTINGS

Almost all compilers make provision for listing source programs when they are compiled. A listing enables the programmer to verify that the program was input correctly, provides a framework for error messages and diagnostics, and serves as an essential component of documentation. It rarely makes sense to suppress the source listing.

A good listing contains more than just the source text. It must establish a co-ordinate system within the program by which program elements can be identified, both for human communication and for association with (compile-time and run-time) error messages. It is common to number either every line or every statement of a program; some compilers use an additional co-ordinate (e.g., code location) to simplify run-time messages. Other useful information (particularly measures of cost) may be included in any extra space remaining on each line.

Much time is spent reading listings (although perhaps not as much as should be). A corresponding amount of thought should be given to their layout and appearance. Few compiler-writers can agree on precise formatting rules, but any user can spot the difference between a haphazard listing and a well-considered one. One non-obvious point learned from hard experience is that the part of each line containing source text should be separated from the rest by visible delimiters, avoiding visual ambiguity.

The placement of source text within the available space can substantially affect the readability and understandability of programs. Careful programmers use blank lines and spaces to improve readability by making visual structure reflect logical structure. The key to effective "paragraphing" is the consistent application of simple rules (e.g., indent the body of each nested structure, such as a for loop or a begin block). Various paragraphing styles have evolved, many of them algorithmically expressable. Automatic paragraphing of source listings is now becoming a popular compiler option. There are many reasons for doing this:

- it relieves the programmer of a tedious chore
- it simplifies the process of modifying programs (since the source deck need not be re-paragraphed)
- it assures an exact correspondence between logical and visual structure, not subject to programmer error
- it ensures a uniform paragraphing style within a group
- it allows programs to be automatically reformatted to new page sizes (e.g., for publication)
- it allows the consistent use of extended character sets (e.g., upper and lower

```
context Portion_of_the_timer_manager;

     type Timer_interrupt_type = (Calender_clock_update_alarm,
          Cpu_scheduler_alarm,Primitive_manager_calender_clock_alarm,
          Flush_interrupt);

     type Alarm_type = (End_of_time_slice, Run_clock_alarm);

     type Timer_element =
          record
               pointer to Process_or_capability_description_type (
                    Process),
               pointer to Timer_element (Next_element),
               Alarm_type (Alarm_reason),
               Timer_units (Time)
          end;

     /*        Insert - - this puts element in its correct place in the *
      * queue and adjusts the interval.                                 */

     macro Insert(Element);

          begin
               open Element@;-

               if Time < First_timer_element@.Time;
                    then:  Next_element := First_timer_element;
                         First_timer_element := Element;
                         Next_element@.Time := Next_element@.Time - Time;
                    else:

                         begin
                              declare
                                   pointer to Timer_element (E);
                              E := First_timer_element;
                              Time := Time - E@.Time;

                              cycle

                              .... exit when Time < E@.Next_element@.Time
                                        | E@.Next_element = End of queue;
                                   E := E@.Next_element;
                                   Time := Time - E@.Time
                              end;

                              Next_element := E@.Next_element;
                              E@.Next_element := Element
                         end;

                         if Next_element    = End_of_queue;
                              then: Next_element@.Time :=
                                   Next_element@.Time - Time
                         end

               end

          end
     end macro;

-|-
```

Figure 1.2.1   An Automatically Paragraphed Listing

case) available on printers, but not keypunches, to add further visual structure Figure 1.2.1. shows the output of a typical automatic paragrapher. Further details may be found in Gordon [1974].

Listings of target programs are not needed nearly as frequently as source listings. They are sometimes demanded by users who do not really trust their compilers, but their most common use is to assist the compiler-writer himself in debugging or maintaining the compiler. The provision of a capability to list the target code generated for specified sections of source code is thus a good investment, but the emphasis should be on simplicity and clarity, rather than on the provision of elaborate features. Figure 1.2.2 shows one possible style.

1.3. SUMMARIES

Although a source listing contains theoretically complete information about a program, it is generally useful to collect some information in a more compact form. The compiler's symbol table is the repository of much of this information, and provision should be made to print it in a convenient form, either at the end of compilation, or at the end of each major unit (e.g., procedure). For ease of access, it should be sorted alphabetically, and for each symbol should indicate at least:

- its type and size
- the co-ordinate where it was defined (declared)
- the number of times it was used

and possibly:

- the co-ordinates of all uses of the symbol (a cross-reference table)
- the (absolute or relative) memory address associated with the symbol (a memory map).

The compiler should indicate the memory requirement (code, and, if possible, data) for each major program unit. Depending on the language and the environment, various other statistics gathered about the program may be helpful. Finally, the number of warnings and error messages printed must be displayed in a conspicuous place, so that the programmer can easily verify that he has not missed any. (If no errors are detected, do not make the mistake of telling the programmer that his program does not contain any errors.)

Compilers should also collect statistics on their own performance. A few lines at the end of compilation indicating the amount of time used by major compiler components, the size of various tables, etc., will not only warn the user who strains various compiler limits, but will also guide the person who wishes to optimize compiler performance.

```
X P L   COMPILATION -- STANFORD UNIVERSITY -- XCOM III VERSION OF MAY 7, 1969.

TODAY IS AUGUST 10, 1969.

 1  |                                                           |  1286
 2  |   /*  INTERLIST $EMITTED CODE  */                         |  1286
 3  |                                                           |  1286
 4  |    DECLARE I FIXED, J BIT(16), K BIT(8),                  |  1286
 5  |       ALPHA CHARACTER INITIAL('MESSAGE'),                 |  1286
                                             24: DESC = 6, 160
                                             160: CHARACTER = D4
                                             161: CHARACTER = C5
                                             162: CHARACTER = E2
                                             163: CHARACTER = E2
                                             164: CHARACTER = C1
                                             165: CHARACTER = C7
                                             166: CHARACTER = C5
 6  |       BETA (3) BIT(64) ;                                  |  1286
 7  |                                                           |  1286
 8  |    CALL TRACE ; /* BEGIN TRACING */                       |  1286
                                             1286: CODE = STM  1,124(3,11)
                                             1290: CODE = LA   1,12(0,0)
                                             1294: CODE = BALR 12,15
                                             1296: CODE = LM   1,124(3,11)
 9  |    I,J,K = 2 ;                                            | 1300
                                             1300: CODE = LA   1,2(0,0)
                                             1304: CODE = STC  1,1346(0,11)
                                             1308: CODE = STH  1,1344(0,11)
                                             1312: CODE = ST   1,1340(0,11)
10  |    BETA(I) = ALPHA ;                                      | 1316
                                             1316: CODE = L    1,1340(0,11)
                                             1320: CODE = L    2,24(0,13)
                                             1324: CODE = SLL  1,2
                                             1328: CODE = ST   2,2811,13)
```

Figure 1.2.2.   Interlisting Emitted Code.

2. REACTION TO ERRORS

In an ideal world, where neither humans nor machines ever made mistakes, the compiler-writer could limit his attention to correctly translating correct programs (and some naive compiler-writers do so). In reality, however, most of the programs processed by any compiler will be to some degree incorrect - simply because correct programs need not be recompiled - and most compilers themselves contain hidden errors. Unless the compiler-writer has planned for them, some errors may have catastrophic consequences.

2.1 STYLES OF REACTION

We may broadly classify a compiler's response to a particular error into one of six categories.

2.1.1 CRASH OR LOOP

Although every compiler should respond reasonably to any possible input, all too often some unforeseen combination of circumstances will place a compiler in a non-terminating loop or cause it to lose control and be terminated abnormally.

2.1.2. PRODUCE INVALID OUTPUT

Sometimes a compiler will apparently operate correctly, yet produce output that is invalid because of an undetected error. It can be argued that such a reaction is potentially more damaging than one in the previous category, since the user is given no indication that anything is wrong.

2.1.3. QUIT

While it is at least honest, a compiler that quits upon first detecting an error will not be popular with its users. Many runs may be required just to remove trivial keypunching errors from a program. (Some puritans argue that this strict discipline will encourage better habits on the part of programmers, however.)

2.1.4. RECOVER AND CONTINUE CHECKING

A compiler may continue looking for errors even after it has abandoned compilation of a program. In order to do so, however, it must somehow get past the original error in such a way that further problems it reports are likely to be symptoms of new errors.

2.1.5 REPAIR AND CONTINUE COMPILATION

An incorrect program may be transformed into a "similar" but valid program, which is then compiled. If carefully done, such transformations may permit the detection of more errors than simple recovery techniques do, and they may also make it possible to guarantee that some parts of the compiler need only deal with valid programs. However, there is no guarantee that the transformed program represents the user's intent.

2.1.6. "CORRECT"

True error correction - the replacement of incorrect programs by the programs the user intended - is substantially beyond the current state of the art of both language design and compiler design. "Error correcting" compilers to date merely repair and hope. Unfortunately, they may also mislead the user who believes that they have really corrected his errors.

2.2 ENSURING CHOSEN REACTIONS

The less desirable reactions (crash or loop, produce invalid output, quit) may be produced accidentally. Even at this level, however, some planning is required to ensure consistent responses to a variety of errors. Compilers never perform well at recovery or repair unless considerable forethought has gone into the design of their error-handling mechanisms.

A compiler should be a total function. To ensure that no possible input can cause the compiler to loop unboundedly or crash, it is necessary to demonstrate that no module ever loses control, regardless of its inputs, and that every loop either has an a priori bound, or involves the consumption of some input. We may call these properties <u>error immunity</u>.

Compilers should attempt to detect and report as many errors as possible. Two obstacles block our desire to strengthen this to "all errors": 1) Some errors transform valid programs into other valid programs (e.g., replacement of "I := I + 1" by "I := I + I"). Errors are more likely to be detected in languages with high redundancy than in those designed for conciseness. 2) Although "correctness is a compile-time property," some errors manifest themselves only under particular dynamic conditions, which can only be determined by the compiler by simulating complete executions of the program. Fortunately, syntactic errors, to which we will devote a great deal of attention, can all be detected by the compiler. There is no excuse for failing to report a syntactic error.

All error detection is based on redundancy. Thus, the symptom of an error is always an inconsistency between two (or more) pieces of information that are supposed to agree. For recovery, it is sometimes sufficient to ignore the inconsistency. Repair requires that at least one of them be changed to achieve consistency. Correction would require sufficient redundancy to determine (with high probability) which item is in error and what its intended value is.

2.3 *ERROR SOURCES, DETECTORS AND SINKS*

Before we can deal effectively with errors, we need to determine where they come from, where they are noticed, and how they are removed.

An error can enter a program in many different ways. The original specifications for the program may be wrong; the programming may not accurately reflect the design (a "logic error"); the source program may not be what the programmer intended (a "keypunch error"); or the compiler itself may introduce an error by incorrect processing. Errors from different sources may frequently exhibit some of the same symptoms within the compiler, but (except for compiler errors) the compiler's chances of dealing properly with an error increase somewhat as we move down the list, due to the kinds of redundancy available to it.

Explicitly or implicitly, each of the analysis phases of the compiler compares its input with a set of specifications. Since not all inputs are valid, there is a possibility of conflict (error detection). Frequently, if somewhat inaccurately, we name errors by the analyser that detects them. Thus we speak of lexical errors, syntactic errors, and semantic errors.

After detecting and reporting an error, a module may either attempt to repair it (so it is not seen by subsequent modules) or pass it along. Each approach has its problems. If a module is to be truly an error sink, it must ensure that none of the effects of the error it has repaired can propagate. Conversely, if it does not filter out all errors, then all subsequent modules must be prepared to deal reasonably with them (without generating too many further messages). In many compilers, a single error can trigger a whole avalanche of messages on the unsuspecting user; this is very nearly as unacceptable as quitting after the first error.

3. SYNTACTIC ERRORS

Syntactic analysis not only plays a central role in the organisation of compilers, it is also the focal point of error detection and recovery within compilers. Because syntactic specifications are precise, it is possible to develop parsers that accept exactly the specified languages; because they are formal, it is possible to prove that the parsers detect any syntactically invalid program. Typically, syntax provides the most stringent single structure within a programming language; more keypunch errors and coding errors will be caught as violations of the syntactic specifications than by all other tests combined.

Recovery from syntactic errors is particularly difficult (and especially important) because of their non-local effects. The omission of a single begin or the insertion of an extra parenthesis may radically change the interpretation of a large section of the program being compiled. By contrast, errors detected during lexical analysis or semantic analysis frequently have only local effects.

3.1. POINT OF DETECTION

Different parsing techniques will in general respond differently to an error that causes a program to be syntactically invalid. Many ad hoc techniques and some of the older formal methods (e.g., operator precedence) will actually fail to detect any errors at all in suitably chosen gibberish. Other methods (e.g., precedence) guarantee to eventually detect some error in every invalid program, but the point of detection may be arbitrarily delayed.

One of the principal merits of the LL(1) and LR parsing techniques [Chapters 2.B and 2.C] is that they guarantee to report error at the earliest point at which it can be determined that an error has occurred, i.e., before accepting the first symbol that cannot be a valid continuation of the portion of the program that has already been read and partially parsed. This early detection makes it much easier to produce meaningful diagnostic messages, improves the chances of successful recovery, and ensures that later compiler modules never partially process source text that turns out to contain a syntactic error.

3.2. RECOVERY TECHNIQUES

A syntactic error is discovered when the parser can take no further valid parsing actions, given the current state of the parse (the stack) and the current input symbol. Recovery thus requires changing the stack, the input, or both. The changes may take the form of deletions or insertions (a substitution is simply a deletion and an insertion). Various combinations of these four kinds of changes have been used in compilers, and the optimal recovery strategy is still a matter for debate.

Arguments can be advanced against each class of change. Gries [1971], for example, points out that changes to the stack are particularly dangerous, since semantic routines will have been invoked for the parsing actions leading to the current stack, and the parser cannot safely undo or modify the effects of these actions. The argument against deleting source text is that some part of the input will thereby not be checked. Source text inserted for recovery purposes is unlikely to correspond exactly with the programmer's intent, and errors detected in the inserted text during semantic analysis may lead to confusing diagnostics. Neither insertion nor deletion by itself is sufficient to recover from all errors; in particular, a parser that attempts recovery solely by insertion may loop unboundedly on some inputs.

3.3. SYSTEMATIC RECOVERY STRATEGIES

Compilers using ad hoc parsing techniques generally rely on ad hoc recovery techniques, i.e., the compiler-writer attempts to supply a "reasonable" recovery action at each point where an inconsistency may be detected. Compilers using formal parsing techniques are more likely to adopt some overall strategy that is expected to cope "reasonably" with any error encountered. No totally satisfactory strategy has yet been demonstrated (and some of the theoretically more attractive ones have not yet been tried in practical compilers). Each strategy can be opposed by particular counter-examples that cause it to perform badly, and comparisons between strategies often rest on undocumented assumptions about which kinds of errors are really the most common (or the most important to recover well from).

Probably the most widely used strategy is also one of the least effective. Panic mode (the term is due to Graham and Rhodes [1973]) has generally been adopted for its simplicity. It merely involves discarding source text until something "solid," such as a statement delimiter, is found, and then discarding stack entries until something that can validly precede the solid token is found. The only merit (besides simplicity) that this method has is that it avoids infinite looping. Almost all the criticisms of the previous section apply.

Two interesting techniques have been reported for using precedence parsing tables to isolate and recover from syntactic errors. Leinius [1970] tries to find the smallest potential phrase containing the point of error that is required by its context to reduce to some unique non-terminal (e.g., <statement>), and then to make the required replacement. Graham and Rhodes [1973], also use precedence relations to isolate the troublesome phrase, but rely more heavily on its internal "resemblance" to some right hand side of a production in choosing a substitute. Both methods suffer from the delayed error detection capabilities of precedence parsers, and violate the criterion of not modifying the stack.

Gries [1971] has proposed an untested scheme for synthesizing a terminal string that will allow the parse to continue based on the context of the error and the productions of the grammar. If no such string can be found, an input symbol is deleted and the search repeated. It is not clear how effective the strategy would be in general, although it does avoid most of the specific problems mentioned in the previous section.

Several authors [e.g., Aho and Johnson 1974, Leinius 1970, Wirth 1968] have considered augmenting the syntactic description of a language by a number of <u>error productions</u>, describing common errors, so that recovery can be (at least partially) subsumed under normal parsing. For this strategy to be effective, several problems must be dealt with: the compiler-writer must ensure that he has really included enough error productions to cover the common errors; since so many different errors are possible, the error productions may substantially enlarge the grammar (and hence the parser); it is difficult to include error productions without making the grammar ambiguous.

Aho and Johnson [1974] have proposed a strategy for LR parsers that seems to be a promising combination of the error production technique with the error isolation ideas of Leinius. Error productions are restricted to the form A→<u>error</u>, where <u>error</u> is a special terminal symbol and A is a "major" non-terminal. When the LR parser encounters an error, it reports it and then replaces the current input symbol by <u>error</u>. Elements are then discarded from the stack until a state is reached with a parsing action for the symbol <u>error</u>. The parser can then read <u>error</u> and reduce by the corresponding error production. Finally, input is discarded until a symbol that can be read in the new state is found.

3.4. *REPAIR TECHNIQUES*

Some diagnostic compilers, notably PL/C [Conway 1973], do a reasonably successful job of transforming programs containing syntactic errors into programs that are both valid and similar. The PL/C technique is somewhat ad hoc, and is based on its authors' long experience with diagnosing errors in students' programs. This section will describe a similar, but somewhat more systematic, method developed for the SP/k compiler by Holt [1973].

Repair is divided into three levels, each with its own delimiters:

- the program level
- the statement level

- the token level

At the program level, if the logical end of the program is not immediately followed by a job control card, input text is discarded until a job control card is found. If a job control card is found before the logical end of the program, input text is _generated_ (by a process to be explained later) until the logical end of the program is generated.

At the statement level, if a semi-colon (or _then_) is expected, but not found, input text is discarded until a _statement delimiter_ (a semi-colon, _then_, or a statement-starting keyword - such as _begin_ or _else_) is reached. If an unexpected statement delimiter is found in the input, input text is generated to complete the current statement.

At the token level, if the current input token is not permitted by the parsing tables, one input symbol is generated. The current input symbol is also discarded _unless_ the generated symbol is a parenthesis or operator and the current input symbol is an identifier or constant.

Generation is controlled directly by the LL(1) parsing tables. If a number is required, "1" is generated; if an identifier is required,"$NIL" is generated; any other terminal symbol is generated for itself when required. If a non-terminal symbol is required, one of its alternatives (generally the most common, or "default" alternative) is selected. The generation process is continued by the parser until repair has been completed at the controlling level, and normal parsing can be resumed.

Although more sophisticated repair strategies can be designed, Holt's simple strategy works surprisingly well. It frequently makes the obvious "correction," and seldom generates avalanche errors; the source text it produces is generally an adequate diagnostic message.

The following figures summarize observations on SP/k's error repair in 83 student programs containing 102 syntactic errors:

errors	"corrections"	Symptom
33	30	missing or unbalanced parenthesis
13	13	missing _procedure_ or _end_
9	8	missing semi-colon
8	6	misspelled keyword
39	15	miscellaneous
102	72	

4. OTHER ERRORS

4.1 LEXICAL ERRORS

Certain errors can be detected purely by lexical analysis. For example, many languages do not utilize the full character set of the computer on which they run (except, perhaps, in character strings and comments). Thus, the detection of an illegal character can be reported immediately; the usual recovery is either to ignore the offending character, or to treat it as a blank. Some classes of tokens (e.g., numbers, identifiers) are formed of restricted character sets. Characters that are invalid in the current token, and that cannot validly follow the current token, may be detected as invalid, even if they are valid in other contexts.

More generally, each token class has its own formation rules, any violation of which must be reported. Many errors can be classified as "delimiter errors" involving tokens that start and end with particular symbols (e.g., comment and semicolon, "/*" and "*/", quotation marks). Failure to terminate such a token with the appropriate delimiter may cause much of the following program text to be inadvertently absorbed into the token. If the same delimiter (e.g., a quotation mark) is used at both ends, the situation is even worse (e.g., all the remaining program text may be treated as strings, and all the strings as program text). To limit the effects of delimiter errors, some compilers bound the length of such tokens (e.g., by limiting them to one card).

Some compilers attempt to recover on the lexical level from certain errors detected by syntactic or semantic analysis. For example, if the parser expects the next token to be one of a set of keywords, but finds an identifier instead, it may inquire of a special "spelling correction" module whether the identifier is a plausible misspelling of any of those keywords. Similarly, if a semantic routine detects the use of an undeclared identifier as a variable, it may inquire whether the identifier is a plausible misspelling of any declared identifier. Morgan [1970] claims that up to 80% of the spelling errors occurring in student programs may be corrected in this fashion.

4.2. STATIC SEMANTIC ERRORS

The number of errors detected during semantic analysis depends both on how many restrictions are specified syntactically and on the amount of checking deferred until run-time. Some checking that could, in principle, be done syntactically (e.g., type compatibility of operators and operands) is frequently left to the semantic routines, for grammatical simplicity and for superior diagnostic capability. If the size or speed of the compiler is of more concern than the speed of the target program (e.g., in a student-oriented compiler), such checking may even be postponed until run-time. However, it is generally preferable to detect errors at compile-time if possible, since equivalent checking at run-time may be executed thousands or millions of times.

Errors of declaration or scope are generally detected during semantic analysis. The most common error in this class is the use of an undeclared identifier. Unless "spelling correction" (Sec. 4.1.) removes the error, recovery is accomplished by treating the identifier as though it had been declared with the type required by its current context; further error messages for the same identifier may be suppressed by entering it into the symbol table with a special "error entry" flag.

Multiple declaration of an identifier within a single scope is an error that is easily detected while adding the new entry to the symbol table. It is debatable whether a better recovery is obtained by discarding the old declaration or the new one; flagging it as an error entry can suppress further messages.

To be useful, every variable must be assigned a value somewhere in the program, and its value must be used somewhere in the program. By keeping a "set" flag and a "used" flag with each variable in the symbol table, the compiler can warn the user of useless variables, which probably represent errors and certainly represent inefficient use of memory.

The remaining major opportunity for static error detection is the discovery of type incompatabilities between operators and operands (or formal and actual parameters, or variables and expressions to be assigned to them). How effective a compiler can be here depends almost totally on the language being compiled. Some languages (e.g., BCPL and BLISS) have only a single data type, so no incompatabilities can arise. Others (e.g., APL and SNOBOL) have several types, but allow the type of a variable to change dynamically, forcing type-checking to be postponed to run-time. Still others (e.g., PL/I) define automatic transformations among most of their types, replacing error-detection by type-conversion. However, some languages (e.g., Pascal) combine a rich type structure with strong typing; in these languages thorough type-checking will catch the majority of errors that have escaped detection by other means. (Strong typing requires that the type of every variable, expression, parameter, etc. be calculable at compile-time; it excludes such constructs as pointers that are not restricted to pointing at objects of a single type and formal parameters whose type is not specified.) The security gained by strong typing is particularly important in the integration and maintenance of large programs.

4.3. DYNAMICALLY DETECTED ERRORS

Run-time error checking is done for a variety of reasons. Some kinds of errors can only be effectively detected at run-time. Any checking postponed from compile-time must be done at run-time. It may be desirable to include <u>redundant checking</u> to duplicate checks made by the compiler, particularly if the program must function more reliably than the compiler, hardware, and operating system that support it, or if the cost of undetected errors may be high.

In order to perform dynamic checking, extra information associated with the program and/or data must be preserved and checked for consistency. Some kinds of checking (e.g., subscripts vs. array bounds) require very small overheads and should always be performed, while others (e.g., dynamic type checking) are very expensive with current hardware and must be carefully justified to warrant inclusion.

The value of a variable to which no assignment has yet been made is meaningless. Although the compiler can sometimes detect instances of use before definition, in general these errors must be detected dynamically. Ideally, we would associate an extra flag bit with each variable, indicating whether it contained a valid value, and test it on each reference. Some compilers have reduced the storage overhead of this checking by using a single value (e.g., the largest negative number) to represent "uninitialised" (with luck, perhaps it will not generate too many error messages for valid programs!). Other compilers merely initialise all variables to some standard value (e.g., zero), and ignore these errors. And an unfortunately large number of compilers fail to take even this precaution, leaving the user at the mercy of whatever garbage was previously in memory.

In any context, there is a range of acceptable values for an expression, determined by machine restrictions (e.g., word length), language considerations, and programmer-supplied information (e.g., array bounds). A careful compiler can determine that some expressions will always be in range (and hence need not be checked dynamically), and that some will never be (and hence can be reported as errors at compile-time); however, there will remain a residue whose range must be checked at run-time. In some cases, such as arithmetic overflow, the hardware itself may supply both the bounds and the checking, but in general the burden of saving the bounds and producing the checking code falls on the compiler.

Two types of range errors can have such severe consequences that many otherwise uncritical compilers generate checks for them: array subscripts out of bounds, and <u>case</u> (or computed <u>go</u> <u>to</u>) selectors out of range. The former might otherwise allow an arbitrary location in memory to be overwritten, the latter might cause the transfer of control to an arbitrary location; in either case, determining the source of the error from its symptoms may be exceedingly difficult.

Errors that are not caught by any of the techniques mentioned so far are generally called "logical errors," and have as their symptoms either incorrect output or failure to terminate (infinite looping). The compiler can provide little assistance with valid, but incorrect (i.e., not what the programmer intended) output, unless the programmer has supplied additional tests by which the output may be checked. However, some compilers do attempt to detect infinite looping. The problem of whether an arbitrary program will halt is formally undecidable, but as a practical matter these compilers cause execution to be interrupted after some set number of times through any loop without exiting (obviously, it must be possible to increase this bound for particular programs). This technique may be somewhat more effective than simply relying on an execution time limit imposed externally by the operating system.

Since (with current hardware) many useful forms of dynamic checking incur substantial overheads, many compilers allow the user to specify the amount of checking to be performed. Typically, full checking is specified during program debugging, and minimum checking for the production version of this program. Hoare [1970] has criticised this practice on the grounds that it is only undetected errors in the production version that are harmful. (He likens it to the practice of keeping a fire extinguisher in your car at all times, except when it is being used!) Another problem is that the symptoms of subtle errors may disappear or shift when checking code is added or removed. However, the economic argument is frequently compelling.

4.4. LIMIT FAILURES

Every real compiler has a number of limits that may be consequences either of its design or of the finite resources at its disposal. Thus a compiler might be limited to a parse stack depth of 75 and to 500 entries in its symbol table; less reasonably, it might restrict identifiers to 8 (or 31) characters, or the number of blocks in a program to 255. Sensible compiler-writers will attempt to minimize the number of such limits (the ideal is one: total space used by the compiler), and to ensure that the average user never encounters them.

No matter how large any particular compiler limit is, some user, some day, will encounter it. Since the encounter will come near the end of compiling some large program, it is very important that the compiler react sensibly, and report not only that a limit was exceeded, but which one, and what its value is. Thus <u>every</u> place in the compiler where some limit <u>may</u> be exceeded <u>must</u> contain a check against the limit. (A very useful technique that guards against forgetting such checks is to access each limited resource solely by means of a procedure or macro that contains the check.)

The target program will also be run with limited resources, and it is essential that the compiler produce code to check run-time limits wherever they may be exceeded. If storage is allocated dynamically on a stack or heap, every allocation must be preceded by a test to assure that sufficient memory is actually available. Limit checking may be a substantial portion of the overhead in dynamic storage allocation, but the consequences of omitting it are intolerable, since very large programs will malfunction inexplicably.

4.5. *COMPILER SELF-CHECKING*

While the compiler-writer should spare no pains to ensure the correctness of his compiler, he should be eternally suspicious of his own accomplishments. Even modules that can "never" receive incorrect input should be error immune (cf. Sec. 2.2.). It is good policy to include tests throughout the compiler that can never fail while the compiler is functioning properly, to provide early warning of compiler malfunctions. (The worst way to find them is by debugging the target code of correct user programs that fail to execute correctly.) Simple consistency checks (e.g., that input is within range, counters never go negative, stacks are never popped when empty, registers are free at appropriate places) have often uncovered subtle errors that would otherwise have remained unnoticed for months or years.

During compiler development it is wise to "instrument" every major module so that its inputs and outputs can be traced at will. If it contains a major data structure (e.g., the symbol table), provision should also be made for dumping its contents, in a readable format, upon request. These self-diagnostic capabilities should not be removed from the production version of the compiler, since their resource consumption is small unless they are used, and they can be invaluable during maintenance.

5. *ERROR DIAGNOSIS*

It is not sufficient to tell the user that his program contains one or more errors. To a very large extent, a compiler's popularity with its users is determined by its helpfulness in locating and explaining their errors. Since debugging is currently one of the largest single costs of computing, economics indicate that the compiler-writer should invest considerable effort in trying to speed this process.

Good error messages will exhibit a number of characteristics:

- they will be <u>user-directed</u>, reporting problems in terms of what the user has done, not what has happened in the compiler
- they will be <u>source-oriented</u>, rather than containing mysterious internal representations or portions of target text
- they will be as <u>specific</u> as possible

- they will <u>localize the problem</u>
- they will be <u>complete</u>
- they will be <u>readable</u> (in the user's natural language)
- they will be <u>restrained and polite</u>.

The final point is of particular importance. It is all too easy for compiler-writers to forget that the user must be the master, the compiler the servant. The attitude (if not the phraseology) of error messages should always be "Oh worthy master, I have failed to fully understand your intentions," rather than "That does not compute" (or, worse "OC5").

5.1. *LOCATING THE PROBLEM*

The first thing the user must be told about an error is where the inconsistency was detected. Frequently he can detect and correct an error simply from an indication of the line and symbol at which a problem was found, provided that error detection was not too long delayed (cf. Sec. 3.1.). A visible pointer into the line is superior to a numerical or verbal description of the location. Where the problem involves identifiers or symbols, they should be given explicitly, e.g., "OUTPOT not declared," rather than "undeclared indentifier" or "THEN may not follow IF" rather than "illegal symbol pair."

5.2. *DESCRIBING THE SYMPTOM*

One of the hardest things to remember in designing error diagnostics is that you don't <u>know</u> what the error was. Two (or more) pieces of information have been found to be inconsistent, but it cannot be said with certainty where the error lies. The safest strategy is to describe the symptom (the detected inconsistency) as clearly as possible before attempting to make any suggestions about the nature of the error.

Symptoms should be described in a positive fashion wherever possible, e.g., "I expected this or this, but found that," rather than "Missing right parenthesis." If the inconsistency involves information from some other part of the program (e.g., a declaration), that information should be displayed, or at least its co-ordinates given.

5.3. *SUGGESTING CORRECTIONS*

The compiler-writer's knowledge of, and experience with, the language being compiled may indicate that certain kinds of inconsistencies nearly always spring from particular kinds of errors ("characteristic errors"). If he is unable to re-design the language to eliminate these errors, he may nevertheless pass this information on to the user in the form of suggestions for correcting the error. Such ad hoc suggestions correspond to the reflex actions of experienced program advisors, in which a particular message will trigger an automatic query of the form "Have you checked...?"

It is not possible for the compiler-writer to separately anticipate every error that may lead to an inconsistency and prepare a suitable suggestion for it. Particularly in the area of syntactic errors, it is probably better to use some simple algorithm (working on the stack, the input, and the parsing tables) to generate "reasonable" suggestions for a whole class of inconsistencies than to consider each one separately.

A compiler that does careful repair (see Sec. 3.4.) may use the repaired source text as a suggested correction. The repaired text will be valid, and "similar" to the input. Even when it fails to match the user's intent, the repair may be very successful in communicating to the user the nature of the problem and the steps that he should take to remove it. (The repaired text should be listed anyhow, so the user can understand further messages about the revised program.)

5.4. *LOCALISATION OF ERROR PROCESSING*

All messages about errors should be processed by a central module. Failure to observe this principle will have numerous undesirable consequences:

- redundant code will be produced throughout the compiler to handle error messages
- subtle inconsistencies in the treatment of errors and formatting of messages will creep in
- the adaptability of the compiler will suffer, and changes to error-handling will be difficult
- it will not be convenient to collect statistics on the number of error messages, nor to suppress duplicate or excessive messages.

The final point deserves further comment. The user should not be bludgeoned with numerous messages springing from a single error. We have already mentioned some recovery techniques to minimize duplicate messages (e.g., error entries in the symbol table). Some compilers suppress all error messages for a line of source text following the first reported syntactic error. The reason is simple: current syntactic error recovery/repair techniques are just not good enough to ensure that further messages really indicate new errors. Most compilers also place a limit on the total number of error messages they will print, since at some point confidence in the whole recovery/repair mechanism breaks down.

5.5. *SYNTHESIS AND PLACEMENT OF MESSAGES*

The error-reporting module should have a standard format for messages that is concise yet distinctive - error messages should stand out on the page. A typical format might include:

- a pointer to the symbol where the inconsistency was noted
- a series of asterisks or other special characters in the margin to catch the eye
- an indication of severity
- a brief description of the symptom
- the count of error messages, and the co-ordinate of the immediately previous message.

Other information would be included depending on the particular symptom, such as

- identifiers, symbols, or types involved
- what was expected by the compiler
- suggested corrections.

Gries [1971] discusses techniques for efficiently synthesizing error messages.

There are two reasonable placements for error messages: in the listing at the point of detection, and following the listing, with the summary information. These placements are frequently, although not necessarily, associated with single-pass and multi-pass compilation, respectively.

For dealing with individual errors, placement of the message at the point of detection is certainly more convenient. The location of the messaoe itself establishes the co-ordinate of the symptom, and it is not necessary to flip back and forth from the listing to the error messages to find the problem. This placement occurs automatically in single-pass compilation unless all messages are saved until the end. If it is desired in multi-pass compilation, listing must be postponed until the completion of analysis, and the error messages collated into the listing.

Error messages that are collected in the summary information are easier to find, and are perhaps less likely to be overlooked. It is essential that all messages dealing with a single statement appear together, and desirable that messages be sorted into the same order as the statements they refer to. For single-pass compilation this simply requires a FIFO buffer in which messages are saved, but with multiple-pass compilation it becomes necessary to collate the messages from the various passes.

5.6. ERROR LOGGING

Of course the user wants to know at the end of compilation how many error messages were issued by the compiler. There are also others who may be concerned with statistics about errors. An instructor in a course may wish to know what kinds of errors his students are currently making, so he can take corrective action. A programming manager might want similar statistics, or statistics about the error-proneness of individual programmers. A language designer could try to eliminate the more common errors, if he had accurate statistics about what they were. Finally, the compiler-writer himself may be able to improve the utility of the compiler by monitoring its response to detected errors.

Different uses may require different kinds of error logging. For some, gross aggregates (e.g., 5,842 severe error messages, 10,914 error messages, 7,231 warning messages) may suffice. Others will need classification by point of detection (e.g., 5,914 lexical errors, 10,231 syntactic errors, 7,842 semantic errors), and still others will need the frequencies of each message. For a detailed study of responses to errors, it may be helpful to save every source statement that caused a message, together with the message, or even to "drain" a copy of every program containing a detected error. (This latter is particularly helpful in evaluating changes in the compiler's response to errors.)

5.7. RUN-TIME DIAGNOSIS

The diagnosis of errors detected at run-time should follow the general principles discussed in previous sections. However, these standards can only be achieved with some forethought, and many otherwise excellent compilers abdicate all responsibility in this domain to an operating system totally unequipped to deal reasonably with run-time errors - the result is a cryptic message and a hexadecimal dump.

The fundamental principle that diagnostics must be given in terms of the source, not the target, program, requires (as a minimum) that the symbol table be available at run-time (although not necessarily in main memory). Dumps, when required, should contain variable names and values in source-language form, and should be selective, working outward from the scope in which the error was detected. [Poole 1973]

Traces should also be based on the source program; ideally, a trace will list the source statements as they are executed, but source co-ordinates provide an acceptable substitute. Since they potentially generate so much output, it is particularly important that traces be selective. Some popular techniques are: letting the programmer dynamically start and stop tracing; tracing only particular sections of the program; tracing only major program elements (e.g., procedure calls and returns); only tracing each statement the first N times it is executed (typically N=2 is adequate); and keeping a ring buffer of trace lines, and printing only those executed immediately preceding the detection of the error.

Another useful tool, both in diagnosing actual errors and in detecting inefficiencies, is the "program profile" - a labelling of each statement by its frequency of execution. Zero-frequency statements are either untested or useless. High-frequency statements are candidates for optimization. Satterthwaite [1972] describes the implementation of profile collection for Algol W, as part of what is undoubtedly the best run-time diagnostic package available today.

References

1. Aho, A.V., Johnson, S.C.: LR parsing. Computing Surveys (to appear 1974).
2. Conway, R.W., Wilcox, T.R.: Design and implementation of a diagnostic compiler for PL/I. Comm ACM 16 169-179 (1973).
3. Gordon, H.E.: Paragraphing computer programs: M.Sc. thesis, University of Toronto expected 1974.
4. Graham, S.L., Rhodes, S.P.: Practical suntactic error recovery in compilers. Conference record of ACM symposium on principles of programming languages, Boston, 52-58 (1973).
5. Gries, D.: Compiler construction for digital computers. John Wiley & Sons, Inc. 1971.
6. Hoare, C.A.R.: The use of high level languages in large program construction. In Turski, W.M. (ed.): Efficient production of large programs. Warszawa 1971
7. Holt, R.C., Wortman, D.B.: Structured subsets of PL/I. Computer Systems Research Group Technical Report CSRG-27. University of Toronto 1973.
8. Koster, C.H.A.: Error reporting, error treatment, and error correction in Algol translation, part 1. Second annual conference of Gesellschaft für Informatik, Karlsruhe 1972.
9. Leinius, R.P.: Error detection and recovery for syntax directed compiler systems. Ph.D. thesis, University of Wisconsin, Madison 1970.
10. Morgan, H.L.: Spelling correction in system programs. Comm ACM 13 90-94 (1970)
11. Poole, P.C.: Debugging and testing. In Bauer, F.L. (ed.): Advanced course on software engineering. Springer-Verlag 1973.
12. Satterthwaite, E.: Debugging tools for high level languages. Computer software - practice and experience 2 (1972).
13. Wirth, N.: A programming language for the 360 computers. Journal ACM 15 37-74 (1968).