References

Bodmer, F. [1944], with Hogben, L. (Ed.). *The Loom of Language.* George Allen and Unwin, London, 7th impr., 1961.

Buxton, J.N. and Randell, B. [1969]. Software Engineering Techniques. NATO Science Committee, Brussels, 1970.

Caracciolo di Forino, A. [1968]. String processing languages and generalized Markov algorithms. In *Symbol Manipulation Languages and Techniques,* D. G. Bobrow (Ed.), North Holland Pub. Co., Amsterdam.

Courant, R. [1961]. In Applied mathematics: what is needed in research and education—a symposium. *SIAM Rev. 4,* 4 (Oct. 1962), 297–320.

Dijkstra, E.W. [1972]. The humble programmer. *Comm. ACM 15,* 10 (Oct. 1972), 859–866.

Gold, M.M. [1969]. Time-sharing and batch processing: an experimental comparison of their values in a problem-solving situation. *Comm. ACM 12,* 5 (May 1969), 249–259.

Hardy, G.H. A Mathematician's Apology. Cambridge U. Press, reprinted 1967. Partially reprinted in *The World of Mathematics,* J.R. Newman (Ed.), Simon and Schuster, New York, 1956, pp. 2027–2038.

Higman, B. [1967]. *A Comparative Study of Programming Languages.* American Elsevier, New York.

Jespersen, O. [1922]. *Language, Its Nature, Development, and Origin.* George Allen and Unwin, London, 10th impr., 1954.

Jespersen, O. [1928]. *An International Language.* George Allen and Unwin, London.

Jespersen, O. [1930]. *Novial Lexike.* George Allen and Unwin, London.

Jespersen, O. [1938]. *En Sprogmands Levned.* Glyndendal, Copenhagen.

Kernighan, B.W. and Plauger, P.J. [1974]. *The Elements of Programming Style.* McGraw-Hill, New York.

Knuth, D.E. [1968–1973]. *The Art of Computer Programming.* Addison-Wesley, Reading, Mass.

Naur, P. [1974a]. *Concise Survey of Computer Methods.* Studentlitteratur, Lund, Sweden, and Petrocelli Books, New York.

Naur, P. [1974b]. What happens during program development—an experimental study. In *Systemering 75,* M. Lundberg and J. Bubenko (Eds.), Studentlitteratur, Lund, Sweden, pp. 269–289.

Ralston, A. [1973]. The future of higher-level languages (in teaching). In *International Computer Symposium 1973,* A. Gunther, B. Levrat, and H. Lipps (Eds.), American Elsevier, New York, 1974, pp. 1–10.

Stone, M. [1961]. The revolution in mathematics. *Amer. Math. Monthly* (Oct. 1961), 715–734.

Strunk, W. Jr., and White, E.B. [1959]. *The Elements of Style.* Macmillan, New York.

Von Neumann, J. [1947]. The mathematician. In *The Works of the Mind,* Heywood and Neff (Eds.), U. of Chicago Press. Also in *The Collected Works of John Von Neumann,* Vol. 1, Princeton U. Press, pp. 1–9; and in *The World of Mathematics,* J.R. Neumann, (Ed.), Simon and Schuster, New York, 1956, pp. 2053–2063.

Weinberg, G.M. [1970]. *PL/I Programming: A Manual of Style.* McGraw-Hill, New York.

Programming Languages

B. Wegbreit Editor

Exception Handling: Issues and a Proposed Notation

John B. Goodenough
SofTech, Inc.

This paper defines exception conditions, discusses the requirements exception handling language features must satisfy, and proposes some new language features for dealing with exceptions in an orderly and reliable way. The proposed language features serve to highlight exception handling issues by showing how deficiencies in current approaches can be remedied.

Key Words and Phrases: multilevel exit, goto statement, error conditions, structured programming, ON conditions, programming languages

CR Categories: 4.22

1. The Nature of Exception Conditions

Of the conditions detected while attempting to perform some operation, *exception conditions* are those brought to the attention of the operation's invoker. The invoker is then permitted (or required) to respond to the condition. Bringing an exception condition to

A version of this paper was presented at the Second ACM Symposium on Principles of Programming Languages, Palo Alto, Calif., Jan. 20–22, 1975.

This work was supported in part by contract DAAA25-74-C0469, Frankford Arsenal, Philadelphia, Pa.

Author's address: SofTech, Inc., 460 Totten Pond Road, Waltham, MA 02154.

an invoker's attention is called *raising an exception.* The invoker's response is called *handling* the exception. Some properties of exception conditions relevant to language features for raising and handling them are:

(a) an exception's full significance is known only outside the detecting operation; the operation is not permitted to determine unilaterally what is to be done after an exception is raised;

(b) the invoker may be permitted to terminate the operation at the point of the exception's detection; the operation may specify that its termination is required;

(c) the invoker controls whether or not a default response to the exception is to be activated; defaults are defined within the operation raising the exception and are executed unless the invoker prevents their execution.

In essence, exceptions permit the user of an operation to extend an operation's domain (the set of inputs for which effects are defined) or its range (the effects obtained when certain inputs are processed). Exceptions permit a user to tailor an operation's results or effects to *his* particular purpose in using the operation. In short, exceptions serve to *generalize* operations, making them usable in a wider variety of contexts than would otherwise be the case. Specifically, exceptions are used:

(a) to permit dealing with an operation's impending or actual failure. Two types of failure are of interest: range failure, and domain failure;

(b) to indicate the significance of a valid result or the circumstances under which it was obtained.

(c) to permit an invoker to *monitor* an operation, e.g. to measure computational progress or to provide additional information and guidance should certain conditions arise.

The value of making this classification of exception uses is the insight it gives into the need for various exception-handling language capabilities. Although it may sometimes be difficult to decide precisely how to classify a given exception, this ambiguity is not crucial for the purposes of this paper.

Range failure occurs when an operation either finds it is unable to satisfy its output assertion (i.e. its criterion for determining when it has produced a valid result), or decides it may not ever be able to satisfy its output assertion. For example, a read operation does not satisfy its output assertion when it finds an end-of-file mark instead of a record to be read; this is a range failure of the first kind. The second type of failure is exemplified by encountering a parity error when attempting to read a record, since in this case, it is uncertain whether repeated attempts to read will or will not eventually be successful. For a numerical algorithm, evidence of divergence is a range failure of the first kind; failure to converge after a certain amount of effort has been expended would be a failure of the second kind.

In general, to deal with range failures, the following capabilities are needed:

(a) The invoker needs the ability to abort the operation; the operation also needs the ability to say it cannot do any more—termination is required. Sometimes as a side-effect of terminating the operation, it is necessary to undo all effects of attempting the operation.

(b) The ability to tell the operation to try again is also required, since this may be a reasonable response in some circumstances (e.g. in the case of reading the bad tape record).

(c) The ability to terminate the operation, returning partial results to the invoker, perhaps together with additional information needed to make sense of the results, is also required. For example, in the case of the bad tape record, the invoker may be able to perform some independent check of the record's validity or he may be able to compensate for any errors found. In short, he may want to modify partial results to make them valid for his purposes. This capability is of obvious utility in increasing the generality of an operation, since the appropriate "fixup" actions (or even the possibility of fixing up the partial results) will vary from one use of the operation to the next.

In short, range failure requires the ability to terminate an operation prematurely (with or without production of partial results and with or without the "undoing" of intermediate results). Range failure also requires the ability to resume the operation when further attempts at completion are deemed reasonable.

Domain failure is a somewhat different type of failure. It occurs when an operation's inputs fail to pass certain tests of acceptability, e.g. the appearance of a letter in a string of digits or the inability to find enough space to satisfy a storage allocation requirement. Domain failure is distinguished from range failure in that domain failure occurs when some *input* assertion is tested and not satisfied, whereas range failure occurs when an output assertion cannot be satisfied.

To deal with domain failures, an invoker must be given enough information about the failure so he can modify the input to satisfy the input criterion if he wishes. If the invoker is unable to fix the problem, he must be permitted to terminate the operation, with or without the operation undoing any "setup" actions taken before the domain failure was detected. The capability an invoker may require to deal with this type of failure is the ability to access the operands of the operation as well as additional information provided from within the operation. The invoker needs this information to help pinpoint in what way the operands are in error.

Exceptions may also be raised to *classify the result* of an operation. In this case, the operation's result satisfies its output assertion, but the invoker needs additional information describing the result before he can give it an appropriate interpretation. For example, addition overflow on many computers produces a valid result as long as the bits of the result are interpreted appropriately. Or an operation processing a list of items

(or reading a file) may return the last item in the list (or file) with an indication that it *is* the last item so the invoker will not attempt to find more items. Note that exceptions of this type are different from range failures because the operation's output assertion is satisfied. Since the output assertion may be satisfied in several ways, however, the invoker needs to know which way it was satisfied so he can use the result appropriately.

Result classification is a type of exception that leads naturally to the use of status variables or return codes (i.e. output parameters whose value designates the type of result produced); there is no need to resume the operation because a valid result has been produced already.

Monitoring is the last class of exception conditions. In this case, the invoker wants to be notified when some condition occurs. This is not because the condition indition indicates a failure or the type of result being produced. Rather it is because he simply wants to keep track of the computation's progress, or the operation may need additional information at certain points and it is too expensive to calculate this information every time the operation is invoked, since the conditions under which it is needed occur only rarely. When a monitoring type of exception is raised, an invoker may wish to terminate an operation; more frequently, the invoker may be *required to resume* the operation because it is not possible or economical to terminate the operation cleanly at every monitoring point. One example of a desirable use of monitoring is in conjunction with an operation for searching through a data structure. Each time an item is found, an exception is raised with an argument identifying the item. The invoker can then decide whether or not he wishes to get the next item. If so, he resumes the operation. If not, he terminates it. Resuming the operation is particularly economical if the operation's state has been preserved by the exception handling mechanism. Then, the search algorithm, e.g. for searching a binary tree, can be written recursively and intermediate results can be made available without unwinding the recursion.

In short, exceptions and exception handling mechanisms are not needed just to deal with errors. They are needed, in general, as a means of conveniently interleaving actions belonging to different levels of abstraction [2–7]. They are not necessarily rarely activated. For example, in their use in dealing with result classification, they might be activated on every invocation of an operation. In their use in monitoring operations and receiving intermediate results, an exception might occur many times for a single invocation of the operation. Exception handling notations that imply a fixed implementation technique are therefore not suited for dealing with the complete range of exception requirements. In this paper I propose a notation that is *neutral* with respect to its implementation, so that exceptions raised by different operations can potentially be implemented differently, depending on their expected frequency and type of use.

2. Previous Exception Handling Techniques

A variety of techniques have been used and suggested for dealing with exception conditions. They all have serious drawbacks of one kind or another, but to place the paper in perspective, it is useful to review other exception handling approaches briefly.

The most well-known and commonly used approach is to supply subroutine calls with extra parameters for dealing with exceptions. Three types of parameters are commonly used.

(a) *Subroutines.* An exception handler is coded as a subroutine and its name is passed as a parameter of a call. This method has been advocated by Parnas [8, 9].

(b) *Labels.* In this case, the handler begins at the statement whose label is passed as a parameter. Control is transferred to the label parameter when an exception is detected.

(c) *Status variables.* An integer valued parameter is assigned a value before returning from a subroutine call; the assigned value indicates whether an exception has occurred, and if so, which one. As a variant of this method, the subroutine may be coded as a function whose returned value indicates whether and what exception has occurred.

Hill [10] has previously analyzed the relative virtues of these methods.

Another set of techniques in effect associates implicit subroutine, label, or status parameters with operations. These techniques are:

(a) *Object-oriented exception handlers.* In this case, a handler subroutine (or label) is associated with an object, and subsequently, control is transferred to the handler when the object is used with certain operations and certain conditions arise. For example, the AED Free Storage Package [11] permits a programmer to establish different areas of storage, called *zones*. When a zone is created, the programmer may specify a subroutine to be called if an allocation request for that zone cannot subsequently be satisfied. Such a subroutine serves as an implicit exception parameter of the storage allocation operator.

(b) *Handler setup calls.* In this case, there exists an operation for associating a handler with an exception that can be raised subsequently by some operation. For example, in a package for formatting program output [12], a subroutine NEW.END can be specified to be called in place of a default subroutine when a line is too long to be printed. The association between the line overflow exception condition and NEW.END is established by executing the call SETEND(NEW.END). NEW.END will now be called when a line overflow is detected by a print operation.

(c) *PL/I ON conditions.* The properties of PL/I ON conditions [13] are complex, but in essence they provide a means of associating a programmer-defined subroutine with an exception condition.

A third set of techniques has been proposed pri-

marily for dealing with range failures. These techniques are based on the idea that recovery from a range failure is impossible; the only sensible action is to try some alternative operation that may succeed in obtaining the desired effect. This leads to a "checkpoint retry" approach in which, when failure is reported, an alternative operation is performed. Specific techniques following this approach are the "backtracking" technique [14] and the recursive cache [15], a proposed hardware concept for making more efficient the activity of restoring an environment after failure has been reported. Hoare has suggested a similar concept [16], written as:

$Q1$ **otherwise** $Q2$

meaning that if $Q1$ fails, then $Q2$ should be performed. Hence, $Q2$ acts as an exception handler for the failure of $Q1$. In Hoare's proposal, the failure of $Q1$ does not imply the effects of $Q1$ are undone before $Q2$ is invoked.

We will not discuss any of these techniques further. Keeping them in mind, however, may be useful in understanding the wide range of circumstances under which exception conditions are useful. Moreover, we will argue later that any of these techniques could be used in implementing our proposed exception handling notation.

3. Exception Handling Requirements and Issues

The remainder of the paper discusses in detail a proposed notation for dealing with exceptions. The topics to be addressed fall into the following classes.

(a) *Association of handlers with invocations of operations.* Since exceptions occur when attempting to perform some operation, one basic issue is how to associate the proper handler with the invocation of a given operation.

(b) *Control flow issues.* These issues concern how to ensure the user and the implementer of an operation agree on whether termination or resumption of an operation is permitted when a particular exception is raised, and how a programmer expresses which of these possibilities is being chosen.

(c) *Default exception handling.* It is useful to provide default handlers for exceptions raised by an operation but not handled by an invoker of the operation. Default exception handling capabilities must be provided in a uniform manner for both language-defined and programmer-defined exceptions.

(d) *Hierarchies of operations and their exceptions.* Exception handling issues that arise from the interaction between an exception raising operation and its immediate invoker are somewhat different from those that arise when an exception is disposed of by an indirect invoker. These issues are sufficiently different to deserve special attention.

One issue not discussed in this paper, but critically important nonetheless, is methods for associating parameters with exceptions. Analysis of this issue has not been completed.

3.1 Associating Exception Handlers With Operations

In discussing the various methods for associating handlers with exceptions raised by operations, the differences between an *operation*, its *invocation*, and its *points of activation* need to be kept in mind. An *operation* is either a subroutine or a language-defined operator like addition. An *invocation* is an attempt to execute the operation. An *activation point* is the place from which an operation is invoked. For example, the loop

```
DO I = 1 TO N;
  CALL F (I*I*I);
  END;
```

specifies N invocations of the operation F and 2N invocations of *. However, there is only one activation point for F and there are two for *.

Handlers associated with exceptions are of two kinds: default and invoker-defined. One distinction between these handler types is that invoker-defined handlers can be different for different operation invocations, but default handlers are the same for every invocation. Another distinction is that default handlers are executed only if a programmer decides not to override them. Examples of default handlers are those defined for certain language-defined exceptions in PL/I, e.g. OVERFLOW. When the OVERFLOW exception is raised, there is a system-defined action that will be performed by default unless some invoker-defined overriding action has been specified. We will discuss default exception handling issues in a separate section. In this section, we are primarily concerned with how invoker-defined handlers are associated with exceptions.

In devising language features for exception handling, a key consideration is that the notation help prevent and detect programmer errors [23]. There are at least three kinds of errors to be guarded against.

(1) Forgetting that an operation can raise a particular exception, and so not giving the exception due consideration.

(2) Associating a handler with the wrong activation point.

(3) Associating a handler with the wrong exception (this is possible only when an operation can raise more than one exception).

The proper way to deal with these error possibilities is to devise a notation that makes compile time detection of errors possible. Therefore, it is reasonable to require:

(a) *explicit declaration* of what exceptions an operation can raise. For example, if F and G are subroutines able to raise exception X, they should be declared as follows:

```
DCL F ENTRY (FIXED) RETURNS (FIXED)[X:...];
DCL G ENTRY (FIXED)[X:...];
```

(b) *static association* of exception handlers with

activation points (i.e. associations defined at compile time rather than at run time).

To satisfy the requirement for static association, handlers are attached to syntactic units containing activation points (see Figure 1). In particular, handlers can be attached to procedure calls:

CALL G(A); [X: **handler-action**]

or to expressions containing operators, e.g.

(A + B) [OVERFLOW: . . .]

More than one handler can be attached to a syntactic unit, e.g.

```
(A/F(B)) [OVERFLOW: . . .
          ZERODIVIDE: . . .
          X: . . . ]
```

Handlers may be associated with statements, e.g.

H = A + B; [SIZE: . . .]

where SIZE is the exception raised when the value of A + B is too large to be represented exactly as the value of H. SIZE is an exception raised by the assignment operator.[1] Handlers may also be attached to groups of statements, e.g. loops:

```
DO WHILE (. . .);
  . . . /*loop body contains READ operations*/
  END; [ENDFILE: . . . ]
```

The notation for associating handlers with loops and subroutine calls is similar to that proposed by Bochmann [20] and Zahn [26].

Handlers may also be attached to procedure definitions, in which case it is useful to define the scope of the procedure to include any handlers attached to it.

A handler is invoked if the exception for which it is defined is raised by an activation point within the handler's *reach*. The *reach* of a handler is the syntactic unit to which it is attached (see Figure 1), excluding any contained syntactic units having a handler definition for the same exception. This is similar to the usual block structure name scope rule, except that here the "blocks" are parenthesized expressions, simple statements, statement groups, loops, and subroutine bodies. Note that the statements comprising a handler definition are not themselves within the reach of the handler; they are within the reach of handlers attached to some containing syntactic unit. For example, in

(A * (B + C) [OVERFLOW: . . .]) [OVERFLOW: . . .]

the reach of the first OVERFLOW handler is the expression (B + C), whereas the reach of the second handler is the total expression, excluding the syntactic units for which an OVERFLOW handler is already

[1] In PL/I, the SIZE exception is not checked for unless the programmer *enables* it. In my notation, providing an override handler for an exception is equivalent to enabling it. If the default handler for SIZE ignores the exception, then invoking the default handler (see Section 3.3) could be interpreted as disabling the exception.

Fig. 1. Syntax describing how exception handlers are associated with syntactic units. Syntactic forms enclosed in large square brackets are optional; a tilde signifies one or more repetitions of the preceding syntactic form; braces enclose alternatives (listed vertically), of which one must be chosen.

```
<function call>      ::= <function name> [(<parameter list>)] [[<handler group>]]

<procedure call>     ::= CALL <procedure name> [(<parameter list>)] ; [[<handler group>]]

<assignment>         ::= <variable> = <expression> ; [[<handler group>]]

<loop>               ::= DO <loop form> ; [<statement> ~] END ; [[<handler group>]]

<statement group>    ::= DO; [<statement> ~] END ; [[<handler group>]]

<quantity>           ::= { <constant>
                         | <variable>
                         | <function call> }

                         { <unary operator> { <quantity>
                                            | (<expression>) }
                         { <quantity>
<expression>         ::= { <expression>  } <binary operator> { <quantity>
                                                             { <expression> }
                         (<expression>) [[<handler group>]] }

<subroutine definition> ::= <label> : PROCEDURE [(<parameter name list>)]
                            [[<exception declaration>]] ; [<statement> ~]
                            END ; [[<handler group>]]

<handler group>      ::= { { <exception name> : } ~ <statement> ~ } ~
```

defined (i.e. (B + C) and any such units contained within the first OVERFLOW handler's definition).

The definition of a handler's reach makes it possible to associate a handler with more than one activation point. This makes programs more readable than if a handler could be associated with only one activation point. For example, writing

(A * (B + C)) [OVERFLOW: . . .]

is the appropriate way to say that overflow is to be handled the same way whether raised by the add or multiply operator. When several activation points are to have the same handler, it is useful to be able to substitute a single handler definition for repeated identical definitions. The proposed notation permits this consolidation of unchanging information.

The ability to associate a handler with several activation points is useful, but it does make it possible to inadvertently associate the wrong handler with an activation point. For example, consider the following statements:

```
H = F(A) + F(B) [X: . . . ];
I = (F(C) + F(B)) [X: . . . ];
```

In the statement assigning to H, no handler is given for exception X raised by F(A). Even if the statement assigning to H is nested in a context that provides a handler for X, e.g.

```
DO;
  H = F(A) + F(B) [X: . . . ];
  I = (F(C) + F(B)) [X: . . . ];
  . . .
  END; [X: . . . ]
```

there is no way to check the real intent of the programmer! Did he really mean to write the first statement,

or did he mean to write

H = (F(A) + F(B)) [X: . . .];

The advantage of being able to associate a single handler with several activation points probably more than offsets the possibility of making this sort of error.

Exceptions raised by a subroutine call or language-defined operator are called *implicitly raised exceptions*. If an activation point for an implicitly raised exception does not lie within the reach of a handler for that exception, the program is in error unless the exception is a default exception (see Section 3.3), in which case the exception's default handler is automatically invoked. Note that in contrast to PL/I, exceptions raised implicitly within a subroutine are not automatically passed to the subroutine's invoker; if they are to be passed one level higher, then they must be raised explicitly, using one of the commands given in the next section.

Static associational methods, together with declarations of what exceptions programmer-defined operations can raise, make it possible for a compiler to detect failure to deal with every exception an operation can raise. Such compile-time checks for errors are not, in general, possible with run-time associational methods like PL/I ON conditions. Of course, run-time methods permit different handlers to be associated with the *same* activation point, but this increased power is not particularly useful (an example is given in [1]); whatever effects can be achieved with run-time association can be achieved by the proposed static method with equal convenience and with increased reliability.

It is important that methods for associating handlers with exceptions should help prevent and detect errors. It is also important to deal *uniformly* with exceptions raised by language-defined operations (e.g. addition) and programmer-defined operations (i.e. subroutines). The proposed method obviously is uniform in this respect, unlike the use of subroutines, labels, or status variables passed as parameters.

Another important requirement is the efficiency with which exceptions can be handled. It has been argued [17, 18] that the cost of setting up an exception handler association should be low relative to the cost of activating the handler, since exceptions occur only rarely. This argument equates exceptions with operation failures, and I have explicitly taken a broader view. Sometimes it is reasonable to raise an exception every time an operation is invoked, and perhaps, several times per invocation. None of the existing exception handling methods mentioned earlier is optimally efficient for dealing with every type of exception condition. Although arguments can be made about the relative efficiency of the various techniques for setting up and handling an exception (see [1]), the key advantage of the method proposed in this paper is that *any of the various implementation techniques* can be used, and in a properly supported system, a programmer will be able to specify with a compiler directive what method of implementation will be used for particular exceptions. In this way, a programmer can control implementation efficiency without having to rewrite his program. The *implementation neutrality*[2] of the proposed notation is one of its most attractive properties, although admittedly, there are few or no systems in existence that would permit this property to be exploited. Nonetheless, the notation presents an opportunity that is foreclosed by the other methods.

3.2 Control Flow Issues

There are basically two types of control flow issues:

(a) issues relevant to raising an exception and activating a handler for it;

(b) issues relevant to leaving an exception handler, so the "normal" flow of control can be continued.

ESCAPE, SIGNAL, and NOTIFY. The various situations in which exceptions are useful present different possibilities for resuming or terminating an operation. To guard against error, each exception should have its resumption or termination constraints specified explicitly and in a way that permits violations of these constraints to be detected at compile time. For this reason we divide exceptions into three types:

(a) ESCAPE exceptions, which *require* termination of the operation raising the exception;

(b) NOTIFY exceptions, which *forbid* termination of the operation raising the exception and require its resumption after the handler has completed its actions; and

(c) SIGNAL exceptions, which permit the operation raising the exception to be either terminated or resumed at the handler's discretion.

Every exception must be declared to be one of these types, and it is an error if a declared exception is not treated in accordance with its type declaration. The result of distinguishing these control flow possibilities in the actual text of a program is greater clarity in dealing with exceptions and in understanding programs.

All existing exception handling techniques are deficient either in not making control flow constraints sufficiently explicit or in not being able to handle the entire spectrum of control flow possibilities inherent in the different reasons for raising an exception. For example, PL/I permits an invoker to resume or terminate *any* operation raising an exception, but it does not permit expressing constraints about whether termination or resumption is permitted or required. Other standard methods are explicit about their control flow options, but they are not flexible enough to handle all the possibilities. For example, status variables are appropriate for dealing with result classification exceptions, but since the status value methods require terminating an operation before the status value can be processed, this technique is not suitable for notifying or signaling.

[2] The concept of an implementation-neutral notation is the essence of what has been called elsewhere [24, 25] the uniform referent concept.

Passing exception handling subroutines as parameters is really suitable only for the NOTIFY control flow discipline. (Note that the value of distinguishing NOTIFY exceptions is that if an operation is guaranteed to be resumed after an exception is handled, the implementor of the operation need not spend any effort or space to take care of the possibility of being terminated.)

The exception handling method being proposed in this paper permits an invoker to terminate or resume operations and it makes control flow constraints explicit. For example, with the proposed method, a subroutine F taking a single argument and raising exceptions X and Y must be declared as follows if X is an ESCAPE exception (i.e. if the operation of F cannot be continued after X is raised) and if Y is a SIGNAL exception (i.e. if F's operation can be terminated or resumed by the handler for Y):

```
DCL F ENTRY(FIXED) [X: ESCAPE, Y: SIGNAL];
```

The definition of F is similarly required to specify what exceptions can be raised and their type:

```
F:  PROCEDURE(AA) [X: ESCAPE, Y: SIGNAL];
      DCL AA FIXED;
```

To raise the exceptions X or Y from within F, a programmer must write either ESCAPE X or SIGNAL Y, e.g.

```
DO WHILE "exception Y should be raised";
  SIGNAL Y;
  END;
IF "exception X should be raised";
  THEN ESCAPE X;
```

Writing ESCAPE Y or SIGNAL X is an error. A compiler can verify that the handler for X does not attempt to resume F, and that the implementer of F does not make incorrect assumptions about whether F can or cannot be resumed after raising exceptions X or Y. NOTIFY exceptions are defined and raised similarly, and the NOTIFY control flow constraints can similarly be enforced by a compiler.

Exceptions raised by an ESCAPE, SIGNAL, or NOTIFY command are called *explicitly raised exceptions*. If an exception raised by an ESCAPE, SIGNAL, or NOTIFY command lies within the reach of a handler for that exception, then that handler is executed. If these commands do not lie within the reach of such a handler *and* if the subroutine containing the commands is permitted to raise this exception, the exception is raised within the subroutine's invoker. It is an error to explicitly raise an exception that either does not lie within the reach of a handler for it or cannot be passed to a subroutine's invoker.

Exceptions can be raised by operations executed within *handlers*, since otherwise the actions permitted within handlers would be too restricted. For example, it is convenient to be able to write:

```
CALL F; [X: CALL G; [Y: . . . ESCAPE Z;] . . . ]
```

meaning that if X is raised, G will be called, and if G fails by raising Y, then the handler for X (and Y) will be exited by raising the exception Z. Note that since the reach of a handler does not include the statements comprising the handler, it is possible to substitute ESCAPE X for ESCAPE Z without recursively invoking the handler for X. Similarly, if Z were a handler associated with the invocation of F, e.g.

```
CALL F; [X: CALL G; [Y: . . . ESCAPE Z;] . . .
        Z: . . . ]
```

the handler for Z is not invoked by the ESCAPE Z statement, since the reach of Z's handler does not include the handler for X or Y. Hence ESCAPE Z must invoke some other handler for Z.

The ability to pass exceptions up to an enclosing context is convenient because it is quite natural to deal with an exception first locally and then more globally [19]. For example, it is often convenient to write something like this:

```
H: PROCEDURE [X: ESCAPE];
    . . .
    B = F(A); [X: . . . ESCAPE X;]
    . . .
    END; [X: . . . ESCAPE X;]  .
```

meaning that the exception X is first dealt with locally, and when either no fixup is deemed possible or no further action using local context is useful, the exception is passed on up to the next handler, in this case one attached to the procedure body. When this more global action is completed, the exception is passed up to the procedure's invoker. This is often quite natural and is more convenient than having to invent different exception names just to prevent naming conflicts.

The ESCAPE statement serves naturally as a multilevel loop exiting method, e.g.

```
DO WHILE ( . . . );
  DO WHILE ( . . . );
    . . .
    IF . . .
      THEN ESCAPE X;
      ELSE . . .
  END;
END; [X: . . . ]
```

When X is raised, both the inner and outer loops will be terminated. Similarly, exceptions raised by *operation* invocations within a loop (instead of with an ESCAPE statement) can cause exiting from the loop, e.g. if F is able to raise the ESCAPE exception X, then we could write:

```
DO WHILE ( . . . );
  . . .
  B = F(A); [X: ESCAPE X;]
  . . .
  C = F(A); [X: ESCAPE X;]
  . . .
END; [X: . . . ]
```

The ESCAPE X statement written with each invocation

of F shows explicitly that X is not handled within the loop, but instead is passed to a containing context. It would certainly be possible to permit F to be invoked without explicitly attaching a handler to F(A), but for clarity, it seems more reasonable to explicitly declare disinterest in dealing with X locally. This should make the program more understandable to readers, who will be reminded thereby that the exception X is raised by F and not by other operations invoked in the loop.

The EXIT Command. In the exception handling approach being proposed in this paper, the last executed statement in a handler must explicitly state whether the operation raising the exception is to be terminated or resumed. Termination of an operation raising an exception is expressed by having the flow of control permanently leave a handler, either by raising an ESCAPE type exception, by executing an EXIT statement, or by executing a RETURN statement (which causes not only the handler to be exited, but returns control from the subroutine containing the handler).

The EXIT statement has a valued and nonvalued form. The valued form, EXIT (value), is used to specify a value for an expression, function call, or function subroutine to which the handler being EXITed is attached For example, suppose a programmer desires to check whether the sum of A and B exceeds C; note that if A and B are known to be nonnegative in value and if their sum overflows, the sum does exceed C. Using the proposed notation, the programmer could write:

```
IF (A + B > C) [OVERFLOW: EXIT(TRUE);]
  THEN . . .
```

Similarly, −1 will be assigned to D if the multiplication *or* the addition overflows in the following expression:

```
D = (A * (B + C)) [OVERFLOW: EXIT (−1);];
```

This example shows that EXIT terminates the whole syntactic unit to which the handler is attached as well as the operation raising the exception.

When a valued EXIT statement terminates a handler attached to a function body, the effect is the same as a normal return, e.g.

```
F:  PROCEDURE( . . . ) RETURNS(FIXED);
      . . .
      END F; [OVERFLOW: EXIT (−1);]
```

F returns the value −1 if this OVERFLOW handler is executed. It would be equally valid to write RETURN(−1) in place of EXIT(−1).

The nonvalued form of EXIT statement terminates execution of the *statement* containing the exception-raising operation. For example, in the statement assigning to D, above, if EXIT(−1) were replaced by EXIT, no assignment to D would be performed if OVERFLOW is raised; the next statement executed would be the one following the example statement. Similarly, in the following program fragment, if exception A is raised, the next statement executed will be the one following the entire conditional statement:

```
IF Q[A: EXIT;]
  THEN . . .
  ELSE . . .
```

An EXIT from a handler attached to a procedure body is equivalent to a normal return from the procedure.

In my original paper [1], I defined a third variant of the EXIT statement in which a programmer could specify the name of the operation being terminated when a handler is exited. But this additional EXIT command complexity provides no new expressive power, and only makes the EXIT statement more complex to implement and understand. The valued and nonvalued EXIT statement forms are sufficient.

The RESUME Command. The RESUME command (also proposed in [18]) is used to return control to a subroutine or language-defined operation raising an exception. RESUME returns control to the statement following the SIGNAL or NOTIFY command raising the exception. RESUME must be used to leave a handler for a NOTIFY type of exception, and may be used to leave a handler for a SIGNALED exception.[3]

To illustrate the use of RESUME, consider a subroutine SCAN(P, V) that takes a pointer P to a data structure, SIGNALing the exception VALUE once for each item in the data structure. The value of the item is stored in V when VALUE is raised.[4] To find the sum of all elements in the structure, one could write:

```
SUM = 0;
CALL SCAN(P, V); [VALUE: SUM = SUM + V;
                          RESUME;]
```

SCAN returns when all values have been found. If P is assumed to point to an array of positive values, with a negative value indicating the last value, SCAN might be implemented as follows:

```
SCAN:  PROCEDURE(P, V) [VALUE: SIGNAL];
       DCL A (100) BASED (P) FIXED;
       DCL V FIXED;
       DCL I FIXED;
         DO I = 1 TO 100 WHILE (A(I) > 0);
            V = A(I);
            SIGNAL VALUE;
            END;
       END SCAN;
```

Note that control will return to the statement following SIGNAL VALUE when the invoker of SCAN executes RESUME.

The ENDED Exception. If SCAN is used to search for a value, X, where X is to be inserted into the scanned structure if it is not already present, the following state-

[3] A.I. Wasserman [private communication] has suggested that a handler sometimes should re-initiate the exception-raising operation rather than RESUME it. The ability to retry an operation is not conveniently supported by my proposed notation.

[4] The manner in which VALUE is supplied with a parameter is not satisfactory in several ways, but it suffices for the purposes of this section and the next.

ments could be written:

```
DO;
  CALL SCAN(P, V);
  CALL INSERT(P, X);                /*X not found*/
END; [VALUE: IF V = X
                THEN EXIT;          /*X found*/
                ELSE RESUME;]       /*keep looking*/
```

A rather elaborate control structure is needed just to insure INSERT(P, X) is invoked only under the appropriate circumstances. To avoid such circumlocutions, it is useful to assume that an exception called ENDED is raised whenever an operation terminates normally [20] (see Figure 2). By taking advantage of this exception, we can write in place of the previous program:

```
CALL SCAN(P, V); [VALUE: IF V = X
                            THEN EXIT;
                            ELSE RESUME;
                  ENDED: CALL INSERT(P, X);
                         EXIT;]
```

ENDED is an ESCAPE exception, i.e. it is an error to attempt to RESUME the operation raising ENDED. The ENDED exception is raised when execution of a syntactic unit having an ENDED handler is completed normally. ENDED cannot be raised explicitly with the ESCAPE statement and is not raised if evaluation of a syntactic unit is terminated because some other exception has been raised.

ENDED can usefully be attached to loops as well as to subroutine calls. For example, consider a program that searches an array, A, containing M distinct values in order to find where a given value X occurs; if X is not present in A, it is to be inserted. In addition, there is another array B, where B(I) is to equal the number of times the value A(I) has been searched for. This program can be written as follows (cf. [21, pp. 266, 267, 277 and 26, p. 283]):

```
DO I = 1 TO M;
  IF A(I) = X
    THEN ESCAPE FOUND;
END; [FOUND: B(I) = B(I) + 1;
             EXIT;
      ENDED: M = M + 1;
             A(M) = X;
             B(M) = 1;
             EXIT; ]
```

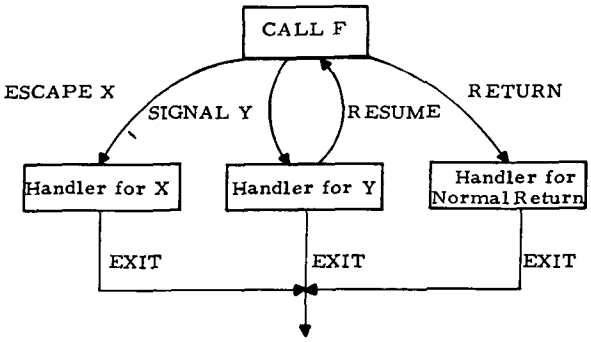
The exception ENDED is, in this case, raised if and only if FOUND is not raised.

The CLEANUP Exception. When an operation is not RESUMEd from a SIGNALed exception handler, it may be necessary first to release certain areas of storage, close files, restore data structures to a consistent state, etc. [17]. Since such "cleanup" actions are only required for SIGNALed exceptions, it is natural to treat CLEANUP as an exception associated with the SIGNAL operation, viz.:

```
SIGNAL Y; [CLEANUP: . . . ]
```

For reasons to be explained later, CLEANUP cannot be attached to any syntactic unit other than a SIGNAL statement. Moreover, a CLEANUP exception can only be raised *implicitly*; it is an error to write ESCAPE CLEANUP, SIGNAL CLEANUP, or NOTIFY CLEANUP.

Fig. 2. The need for ENDED. Before processing can resume independently of whether F raised any exceptions or terminated normally, it may be necessary to perform some actions necessary *only* when the operation terminates normally.



CLEANUP handlers are executed beginning with the handler (if any) attached to the least recently executed SIGNAL statement in a chain of SIGNAL statements. Note that the most recently executed SIGNAL statement in this chain is the one whose handler is causing operations to be terminated. To see the sequence of CLEANUP handler executions, consider the following program:

```
G:  PROCEDURE [LAST: SIGNAL, Z: ESCAPE];
      DCL H ENTRY [FIRST: SIGNAL];
        . . .
        CALL H; [FIRST: SIGNAL SEC; [CLEANUP: . . . ]
                        RESUME;]
        . . .
      END G; [SEC: SIGNAL LAST; [CLEANUP: . . . ]
                     RESUME;]
```

Note that if H raises the exception FIRST, SEC will be raised within G, and then LAST will be SIGNALed to G's invoker. If G's invoker decides to terminate G, the CLEANUP handlers attached to SIGNAL FIRST (in H), SIGNAL SEC, and SIGNAL LAST will be executed in the order FIRST, SEC, LAST. This sequence of CLEANUP executions would be the same, of course, even if exceptions FIRST, SEC, and LAST all had the same name.

Note that if the statement ESCAPE Z were executed within G, no cleanup handlers would be invoked because cleanup handlers are only associated with SIGNAL statements. Note also that if EXIT is written in place of the RESUME following SIGNAL LAST, then if G's caller RESUMEs G when LAST is raised, the handler for SEC will be exited. This implies termination of the SIGNAL SEC statement and termination of H. Once H is terminated, the SIGNAL SEC cleanup handler will be executed, but the cleanup handler for SIGNAL LAST will not be invoked, since the handler for LAST (in G's invoker) resumed execution of G.

The specified process of working from the lowest level (innermost) CLEANUP handler to the highest (outermost) is the most natural way of cleaning up a prematurely terminated operation. A CLEANUP handler must be attached only to SIGNAL statements because the sequence of CLEANUP invocations is so closely tied to the sequence of SIGNAL statement executions.

To keep control flow simple, a CLEANUP handler must not raise any exceptions passed outside the handler. (For example, consider the potential complexity if SIGNAL SEC's CLEANUP handler were permitted to raise G's exception Z.) Note that a compiler can determine whether or not any exceptions raised within a CLEANUP handler will be fully processed within the handler, and this constraint is not difficult to enforce.

Statements within a CLEANUP handler are executed in the normal sequence. When there are no more statements to execute, control passes to the next CLEANUP handler to be executed. Note: the EXIT statement is not used to leave a CLEANUP handler, since leaving a CLEANUP handler means execute the next CLEANUP handler and/or terminate an operation, whereas EXITing a normal handler means execute the next statement in the normal control flow.

A shorthand notation involving CLEANUP handlers will be introduced in Section 3.4.

Summary of Control Flow Issues. The issues discussed in this section may be summarized as follows:

(1) *Prevention/detection of programming errors.* The need to classify exceptions according to their control flow constraints (ESCAPE, NOTIFY, SIGNAL);

(2) *Readability/simplicity/uniformity.* The use of ESCAPE-type exceptions to support multilevel loop exits in GOTO-free programming (ESCAPE to handlers associated with loops or statement groups);

(3) *Exception handling requirements.*

(a) The ability to avoid explicit GOTOs to terminate an operation raising an exception: the need to express that a handler is being exited (the nonvalued EXIT statement), and the need to provide a value for a function or expression when exiting a handler (the valued EXIT statement, e.g. EXIT (6));

(b) the need to deal with an exception first locally and then more globally (e.g. ESCAPE X from within a handler for X);

(c) the need to clean up before terminating an operation (the CLEANUP optional exception associated with SIGNAL);

(d) the ability to define an exception handler for normal termination of an operation (ENDED).

3.3 Default Exception Handling

Up to now, our discussion has concentrated on invoker-defined exception handlers. It is very convenient, however, to define *default* handlers for some exceptions. These handlers are executed unless specifically overridden.

Exceptions having default handlers will be called default exceptions. Failure to provide a handler for a default exception is not an error; to the contrary, it is a way of specifying that the default handler is to be executed.

Most exception handling methods do not satisfy requirements for default exceptions very well. These requirements are:

(1) *Declaration of default exceptions.* To help prevent and detect mistakes in using exceptions, exceptions having default handlers should be declared to have them. Only if the existence of a default handler is made known to a compiler can exception handling errors be detected. For example, failure to provide an invoker-defined handler is an error only for nondefault exceptions.

(2) *Programmer-defined default handlers.* It should be possible to establish default exceptions and handlers for programmer-defined subroutine packages; the concept of a default exception should not be limited to system-defined operations (addition, I/O, etc.)

(3) *Uniformity.* The methods for overriding or invoking default exception handlers should be the same for both system-defined and programmer-defined default exceptions.

(4) *Explicit invocation of default handlers.* A programmer must be able to specify explicitly as well as implicitly whether a default handler is to be invoked or overridden, or whether the decision is to be made by a higher level handler.

Default exceptions are notationally essential. They make operation invocations more concise and readable, and make programming more convenient.

Methods for invoking default handlers are provided in standard PL/I only for language-defined exceptions like OVERFLOW. For such exceptions, a programmer can insure that a default handler will be invoked at certain activation points by writing

```
ON condition-name SYSTEM;
```

A corresponding mechanism is not provided for programmer-defined exceptions. MULTICS supports a more elaborate set of facilities for defining and invoking default handlers (see [1] for a brief discussion). But neither standard PL/I nor MULTICS PL/I provide completely suitable default exception facilities.

In extending the proposed exception handling notation to deal with defaults, the first issue is to distinguish those exceptions for which default handlers exist [requirement (1) above]. Then a compiler can enforce the rule that absence of a handler is an error unless the exception has a default handler. I propose that default exceptions be distinguished by declaring them to have the attribute "OPTIONAL", since it is optional for the programmer to provide a handler for these exceptions. For example,

```
DCL F[X: ESCAPE, Z: SIGNAL OPTIONAL];
```

means a handler is required for X but not for Z. Note that ESCAPE exceptions cannot, by their very nature, have default handlers.

Fig. 3. Program showing how programmer-defined and language-defined default exceptions are treated uniformly.

```
G:  PROCEDURE (...) [X:OVERFLOW: OPTIONAL SIGNAL];
    DCL F ENTRY [X: OPTIONAL SIGNAL];
      ...
      CALL F;[X: ...
                 EXIT;]                  /* default handler overridden; F is terminated */
      CALL F;                            /* default handler invoked implicitly */
      CALL F;[X: RESUME(DEFAULT);]       /* default handler invoked explicitly */
      A = B + C;[OVERFLOW: ...
                           EXIT;]        /* default handler overridden; assignment not performed */
      A = B + C;                         /* overflow default handler invoked implicitly */
      A = B + C;[OVERFLOW: RESUME(DEFAULT);]  /* overflow default handler invoked explicitly */
    END G;
```

In calling F, the default handler for Z will be invoked if the call does not lie in the reach of any handler for Z. For example,

CALL F; [X: . . .]

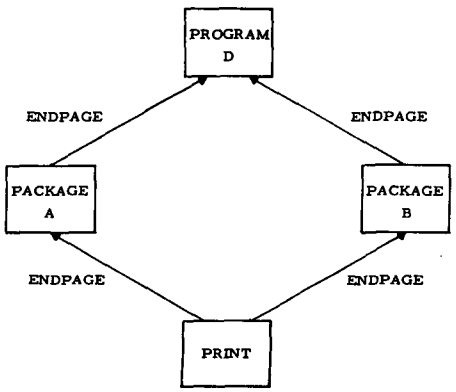
will cause the default handler for Z to be invoked if there is no handler for Z attached to a syntactic unit containing this call statement. Note that in contrast to PL/I, the search for a default handler does not automatically extend beyond the subroutine in which the exception is raised implicitly. This is consistent with the principle of being explicit about important connections between modules. Something as important as an exception condition should not be passed across subroutine boundaries except by explicit command.

An invoker should be permitted to invoke a default handler explicitly as well as implicitly. (It is always dangerous to provide only implicit ways of invoking some capability.) To permit this, we add an option to the RESUME statement. By writing RESUME(DEFAULT) instead of RESUME, a programmer specifies that the default handler for an exception is to be invoked. For example, the call to F could be written:

```
CALL F; [X: . . .
         Z: RESUME(DEFAULT);]
```

to show explicitly that Z's default handler is to be invoked. RESUME(DEFAULT) is analogous to ESCAPE except that it directs control flow downward to a default handler instead of upward to an invoker-defined handler. RESUME(DEFAULT) may be written only within a handler for a default exception.

Fig. 4. An example of multiple default handlers.



The proposed notation for giving an exception a default handler is illustrated below. This is what the implementer of subroutine F would write to define the default handler for Z:

```
SIGNAL Z; [DEFAULT: . . ./*the default handler */
           CLEANUP: . . .
           ENDED: . . . ] /*invoked if default is overridden */
```

In short, DEFAULT is an exception condition associated with the SIGNAL command. This exception is raised if no overriding handler exists for Z or if a handler for Z executes the statement RESUME(DEFAULT). The CLEANUP actions will, of course, be executed only if a handler for Z is exited. The actions associated with ENDED will be executed just in case the default handler for Z is not executed and F is RESUMEd. Note that this is the normal definition of ENDED as applied to the SIGNAL operation. DEFAULT exception handlers can be associated with NOTIFY as well as the SIGNAL operations.

A DEFAULT handler must be associated with every SIGNAL or NOTIFY command used to raise a default exception, e.g. within subroutine F above, it would be an error to write SIGNAL Z without associating a DEFAULT handler with the SIGNAL Z command. DEFAULT, like ENDED, is also an exception that can only be raised implicitly. It is an error to write ESCAPE DEFAULT.

The default exception handling capability described here is not necessarily difficult to implement. In essence RESUME(DEFAULT) is just a GOTO statement referencing a label implicitly associated with a SIGNAL statement.

The proposed default exception handling notation permits programmer-defined and system-defined exceptions to be dealt with uniformly. For example, consider the subroutine in Figure 3. Note that even though G is able to raise exception X and the OVERFLOW exception, these exceptions are not raised explicitly within G. Therefore, they must be disposed of entirely within G; they cannot be passed implicitly to G's invoker.

The RESUME(DEFAULT) command gives higher level programs the ability to substitute default handlers for lower level default handlers. This can be quite useful as is illustrated in Figure 4. The figure shows an application program, PROGRAM D, that makes use of two program packages, A and B, each of which prints ma-

Fig. 5. Substituting a default handler.

```
CALL PRINT; [ENDPAGE: SIGNAL ENDPAGE;
                      [DEFAULT: ...;   /* overrides PRINT default; serves as */
                                EXIT; /*   default for Package A or B       */
                       ENDED:   ...;   /* executed if PROGRAM D provided an  */
                                EXIT;]/*   overriding handler               */
                RESUME;] /* control returns to PRINT */
```

terial using a PRINT utility subroutine. The PRINT utility is assumed to raise the exception ENDPAGE at appropriate points. Instead of having Package A and Package B override the PRINT default handler, they need to signal D to see if D wants to deal with the ENDPAGE exception. If not, A and B will deal with the exception in an appropriate way. As far as D is concerned, the default response for ENDPAGE when using Package A can be different from the default response for Package B. In effect, Package A and B provide their own default response for the ENDPAGE exception if their calls to PRINT are written as shown in Figure 5. Note that the EXIT from the DEFAULT handler in Figure 5 terminates the SIGNAL ENDPAGE operation; it does not terminate the PRINT operation.

The default handler provided by A or B will be executed only if PROGRAM D does not override it. Presumably, the A and B default handlers are different from the PRINT default handler.

It is useful and natural to permit Package A (for example) to resume the PRINT program by invoking PRINT's default handler for ENDPAGE if Program D has invoked A's ENDPAGE default handler. The natural way to express this effect is to write:

```
CALL PRINT;
  [ENDPAGE:  SIGNAL ENDPAGE;
               [DEFAULT: . . . RESUME(DEFAULT);
                ENDED: . . . EXIT;]
             RESUME;]
```

The problem is that RESUME and RESUME(DEFAULT) are commands for resuming the operation with which a handler is associated, and in this case, the DEFAULT handler is associated with the SIGNAL ENDPAGE statement. Resuming the SIGNAL operation does not make sense. The desired effect *can* be achieved, but hardly in a stylish manner:

```
CALL PRINT;
  [ENDPAGE:  DO;
               SIGNAL ENDPAGE;
                 [DEFAULT: . . . EXIT;
                  ENDED: . . . ESCAPE DONE;]
               RESUME(DEFAULT);
                 /*invoke PRINT's default handler for
                  ENDPAGE */
             END; [DONE: EXIT;]
             RESUME;] /*resume PRINT operation */
```

Instead of forcing programmers to write the above sort of program, it is more reasonable to provide an interpretation for RESUME and RESUME(DEFAULT) when they appear in the body of a DEFAULT handler. Specifically, when RESUME and RESUME(DEFAULT) are used as a means of leaving a DEFAULT handler, such statements will be interpreted as though an EXIT statement were executed and then the next statement executed were the RESUME or RESUME(DEFAULT) statements, i.e. the RESUME or RESUME(DEFAULT) commands are interpreted *with respect to the context containing the* DEFAULT *handler*, not with respect to the DEFAULT handler itself.

The system of programs in Figure 4 shows an important distinction between default and nondefault handlers, namely, whether lower- or higher-level programs have the power to pre-empt the handlers at a given level. If a higher-level handler can prevent execution of a lower-level handler, then the lower-level handler is a default handler. If a lower-level handler can prevent execution of a higher-level handler, then the lower-level handler is not a default handler; it is an override handler. The proposed exception handling method gives programmers the ability to permit either a lower- or higher-level program to have the final say in how an exception should be disposed of. Moreover, exception handling actions can percolate up from an operation raising an exception until some invoker is found who is able to deal with the exception, or once the exception has reached the highest level, default handlers can be invoked successively from higher to lower levels until a default handler has successfully dealt with the exception (cf. [22, pp. 189–198]).

The proposed method of dealing with defaults satisfies all default exception handling requirements by extending the previously developed exception handling notation in a uniform way. I assume that language-defined exceptions like OVERFLOW will be defined as default exceptions so a programmer does not have to provide an explicit handler for every operation able to raise such exceptions.

3.4 Exception Handling Hierarchies

Until now, discussion has concentrated mostly on exception handling requirements arising from the relation between an operation raising an exception and its immediate invoker. In this section, some system-wide consequences of the exception handling approach advocated in this paper will be briefly examined.

One set of issues arises from the fact that a subroutine often only mediates between an operation raising an exception and the operation disposing of the exception. For example, suppose operation B is called by operation A. A expects B to raise exceptions X, Y, and Z under certain conditions, but these conditions

Fig. 6. Explanation of the PASS shorthand notation. PASS written without a CLEANUP handler is equivalent to PASS with an empty CLEANUP handler, i.e. [X: PASS;] is equivalent to [X: PASS; [CLEANUP:;]]

Type of exception X	Translation of X: ... PASS; [CLEANUP: ....];
ESCAPE	X: ... . ESCAPE X ;
SIGNAL	X: ... SIGNAL X; [CLEANUP: ....] RESUME ;
OPTIONAL SIGNAL	X: ... SIGNAL X: [CLEANUP: .... DEFAULT: RESUME(DEFAULT);] RESUME;
NOTIFY	X: ... NOTIFY X; RESUME;
OPTIONAL NOTIFY	X: ... NOTIFY X; [DEFAULT: RESUME(DEFAULT);] RESUME;

occur only when B invokes operation C, so C is the routine that actually has responsibility for detecting the exception. B acts merely as a conduit, transmitting exceptions detected by C to A. B could be implemented as follows:

```
B: PROCEDURE [X: SIGNAL,
              Y: OPTIONAL SIGNAL,
              Z: ESCAPE];
  DCL C ENTRY [X: SIGNAL
               Y: OPTIONAL SIGNAL,
               Z: ESCAPE];
  ...
  DO WHILE (...);
     CALL C;
     ...
     END;
  ...
  END B; [X: SIGNAL X;
            RESUME;
          Y: SIGNAL Y;
               [DEFAULT: RESUME(DEFAULT);]
            RESUME;
          Z: ESCAPE Z;]
```

Note that the handlers for X, Y, and Z do nothing except pass these exceptions to B's invoker; there are not even any CLEANUP actions to be performed when B is terminated. Although the basic notation is adequate to express what is to happen, the notation is rather cumbersome, especially if there are many exceptions being passed through to an invoker. It is even more cumbersome if the same cleanup actions are to be performed when C raises any of its exceptions, and B's operation is terminated. For example, we would then have to write:

```
END B; [X: SIGNAL X; [CLEANUP: ...]
          RESUME;
        Y: SIGNAL Y; [CLEANUP: ...
                      DEFAULT:RESUME(DEFAULT);]
          RESUME;
        Z: ... ESCAPE Z;]
```

The three dots in all cases represent the same set of cleanup actions. Obviously there is a need for a shorthand notation to cover this situation, which is not at all uncommon. Before suggesting a shorthand notation, however, note that actions common to several exceptions are not limited to cleanup actions; if X, Y, and Z are all related, a common (possibly null) action in response to them might be required within B, but only B's invoker cares about the finer distinctions implied by X, Y, and Z. Therefore, to deal with actions common to several exceptions, we propose the following notation:

```
END B; [X:Y:Z: ... PASS; [CLEANUP: ...]]
```

where the three dots preceding PASS represent actions to be performed when C raises X, Y, or Z, and the three dots after CLEANUP represent actions to be performed before the operation of B is terminated as a result of passing an exception to B's invoker. The semantics of PASS are specified as follows: for each exception invoking the handler containing PASS (i.e. for exceptions X, Y, and Z in the above case), the handler body is replicated and associated with only one exception; for example, [X: Y: Z: ... PASS; [CLEANUP: ...]] is replaced by:

```
[X: ... PASS; [CLEANUP: ...]
 Y: ... PASS; [CLEANUP: ...]
 Z: ... PASS; [CLEANUP: ...]]
```

Each instance of PASS is then replaced with an appropriate set of commands for passing the exception up to the next handler, according to the type of exception PASS is used with (see Figure 6).

Although the motivation for PASS was the need to pass exceptions to invokers of subroutines, the definition permits use of PASS within a subroutine as well, e.g.

```
B: PROCEDURE...
     ...
     CALL C; [Z: ... PASS;]
     ...
   END B; [X: Y: Z: ... PASS;]
```

This code implies that when C raises Z, some action will be performed in the context of the call to C. Then action common to X, Y, and Z will be performed before the exception C raised is passed to B's invoker.

In short, the ability to treat groups of exceptions identically before passing the exceptions to other handlers is important for notational conciseness, and the need to do so arises naturally when higher level programs need a more detailed breakdown of exceptions than intervening programs.

The insistence that no exceptions be propagated automatically to invokers may seem too burdensome to some, because the automatic propagation of exceptions upwards has at least two apparent advantages:

(a) It makes it easier to add an exception to a system of programs.

(b) Exceptions automatically passed upward are

ultimately passed to the main program's environment, which may be a debugging system or the system's user at an interactive terminal. Considerable flexibility in dealing with exceptions can thereby be provided.

These seeming advantages are more apparent than real. If a programmer adds an exception to a subroutine, and the exception is only disposed of by a subroutine several levels higher in the calling hierarchy, the proposed notation does require that all intervening programs be modified to explicitly pass the new exception through. Although this certainly increases the work of program modification, it is not entirely wasted effort. There are definite advantages in having to examine all intervening programs to see how they are affected by the new exception, and the proposed notation forces such an examination to be carried out.

As for the second requirement, the ability to pass an exception to a main program's environment, it is perfectly reasonable to provide a command for raising an exception in the main program's environment without first passing it up to the main program. Not providing such a command is too restrictive, since programming by stepwise refinement or levels of abstraction implies that top level programs will be unaware of many exceptions raised at lower levels. If a lower level is unable to dispose of some exception and is also unable to pass this exception meaningfully to a higher level, then raising the exception directly in the main program's environment should be possible. Doing so is useful and will drastically reduce the number of exceptions that would otherwise have to be passed up to the main program.

4. Conclusions

In this paper I have discussed the issues posed by exception conditions. I have proposed a new exception handling notation that solves exception handling problems in a more uniform and reliable way than any existing method. The proposed notation succeeds in abolishing nonlocal goto statements to deal with exceptions and subsumes the by-now-familiar device of "break" or "leave" statements for multilevel loop exiting. The proposed notation is not complete, since it does not satisfy the need for parameters associated with exceptions. Nonetheless, the analysis shows that the wide variety of exception handling approaches that exist today can be replaced with a single uniform approach that satisfies the needs of failure, result classification, and monitoring exceptions equally well with reasonable efficiency.

Acknowledgments. I am grateful to M. D. McIlroy for guiding my revision of the original paper [1] by providing many useful suggestions and comments. His observation that my original proposals amounted to attaching handlers to syntactic units was especially helpful. I have also benefitted from comments of the referees.

References

1. Goodenough, J.B. Structured exception handling. Conf. Rec., Second ACM Symp. on Principles of Programming Languages, Palo, Alto, Calif., Jan. 1975, pp. 204–224.
2. Dahl, O-J., Dijkstra, E.W., and Hoare, C.A.R. *Structured Programming*. Academic Press, New York, N.Y., 1972.
3. Woodger, M. On semantic levels in programming. Proc. IFIP Congress 71, North-Holland Pub. Co., Amsterdam, 1972, pp. 402–407.
4. Liskov, B.H. A design methodology for reliable software systems. AFIPS Conf. Proc., Vol. 41, Part 1, 1972 Fall Joint Computer Conference, AFIPS Press, Montvale, N. J., pp. 191–199.
5. Liskov, B.H. The design of the VENUS operating system. *Comm. ACM 15*, 3 (Mar. 1972), 144–149.
6. Dijkstra, E.W. The structure of the 'THE'-multiprogramming system. *Comm. ACM 11*, 5 (May 1968), 341–346.
7. Liskov, B. and Zilles, S. Programming with abstract data types. SIGPLAN Notices (ACM Newsletter) *9*, 4 (Apr. 1974), 50–59.
8. Parnas, D.L. A technique for software module specification with examples. *Comm. ACM 15*, 5 (May 1972), 330–336.
9. Parnas, D.L. Response to detected errors in well-structured programs. Dep. of Computer Sci., Carnegie-Mellon University, Pittsburgh, Pa., July 1972.
10. Hill, I.D. Faults in functions, in ALGOL and FORTRAN. *Computer J. 14*, 3 (March 1972), pp. 315–316.
11. Ross, D.T. The AED free storage package. *Comm. ACM 10*, 8 (August 1967), 481–492.
12. *AED Programmer's Guide*. SofTech, Inc., Waltham, Mass. 1972.
13. Noble, J.M. The control of exceptional conditions in PL/I object programs. Proc. IFIP Congress 68, North-Holland Pub. Co., Amsterdam, 1969. pp. C78–C83.
14. Golomb, S.W. and Baumert, L.D. Backtrack programming. *J. ACM 12*, 4 (Oct. 1965), 516–524.
15. Horning, J.J., Lauer, H.C., Melliar-Smith, P.M. and Randell, B. A program structure for error detection and recovery. In *Operating Systems*, E. Gelenbe and C. Kaiser (Eds.), Springer-Verlag, New York, 1974, pp. 171–187.
16. Hoare, C.A.R. Parallel programming: an axiomatic approach. STAN-CS-73-394, AD769674, Dep. of Computer Sci., Stanford U. Oct. 1973.
17. Organick, E.L. *The MULTICS System: An Examination of Its Structure*. MIT Press, Cambridge, Mass., 1972, 187–216.
18. Lampson, B.W., Mitchell, J.G., and Satterthwaite, E.H. On the transfer of control between contexts. In *Programming Symposium, Lecture Notes in Computer Science, Vol. 19*, B. Robinet (Ed.), Springer-Verlag, New York, 1974, pp. 181–203.
19. Brown, W.S. An operating environment for dynamic-recursive computer programming systems. *Comm. ACM 8*, 6 (June 1965), 371–377.
20. Bochmann, G.V. Multiple exits from a loop without the GOTO. *Comm. ACM 16*, 7 (July 1973), 443–444.
21. Knuth, D.E. Structured programming with goto statements. *Comp. Surv. 6*, 4 (Dec. 1974), 261–301.
22. Elson, M. *Concepts of Programming Languages*. Science Research Associates, Chicago, Ill., 1973.
23. Gannon, J.D. and Horning, J.J. The impact of language design on the production of reliable software. International Conference on Reliable Software, IEEE Cat. No. 75CH0940-7CSR, April 1975, pp. 10–22; also *IEEE Trans. on Software Eng. SE-1, 2* (June 1975), to appear.
24. Ross, D.T. Uniform referents: an essential property for a software engineering language. In *Software Engineering*, J. T. Tou, (Ed.), Vol. 1, Academic Press, New York, 1970, pp. 91–101.
25. Ross, D.T., Goodenough, J.B., and Irvine, C.A. Software engineering: process, principles, and goals. *Computer 8*, 5 (May 1975), 17–27.
26. Zahn, C.T. Jr. A control statement for natural top-down structured programming. In *Programming Symposium, Lecture Notes in Computer Science, Vol. 19*, B. Robinet (Ed.), Springer-Verlag, New York, 1974, pp. 170–180.