Exceptional C: Design and Implementation of the XTC Compiler
Abstract

The development of reliable software requires programs to behave according to their specifications under a wide range of circumstances. In particular, programs should be able to cope with the events that are no part of normal program behaviour, but which could lead to erroneous results, should an appropriate way to deal with them be omitted. These events could be the results of errors, but may also have a number of other causes. In general terms we call such events “exceptional conditions”. An exceptional condition is defined as a state of execution that occurs when an operation is invoked and the action of the operation cannot be completed properly.

An operation that encounters such an exceptional condition at runtime, needs some way of communicating this fact to its invoker. The invoker is then required to deal with the exceptional condition. We will call this “handling the exception”. The best way of doing this is by providing explicit exception handling facilities in the programming language. Exception handling serves to generalise operations, in other words, it makes them useful in a wider variety of circumstances. Existing languages supporting exception handling facilities are either based on a resumption or on a termination model for the operation that detected the exceptional condition, although a combination of both is also possible.

For the Pegasus project it is felt that an exception handling mechanism is needed in the C programming language. We think that such a mechanism will be most useful, if it supports static association of handlers with operations, the possibility to pass arguments of as many different types as possible, and when it incurs only little overhead when no exceptional condition is encountered. Furthermore, we think such a mechanism should be based on the termination model and have language constructs for try blocks, normal, default and finally handlers and an explicit raise statement.

The two approaches towards implementation of an exception handling mechanism, that can be distinguished are “machine independent” and “machine dependent”. For both approaches a compiler has been built, which translates the exception statements to standard C. The former approach can be implemented, either by using the standard library routine setjmp, to save context, or by using nested functions (this is not implemented in a compiler yet). The nonportable solution makes use of a genuine stack unwinding algorithm. The setjmp implementation is completely integrated with gec, which means that it can check on types and absence of handlers. The stack unwinding implementation has only reached the status of precompiler so far. It is, however, able to deal with a slightly more complicated syntax, in which the exception language constructs are treated as ordinary C statements. The stack unwinding algorithm is based on range tables that determine at compile time which handler should be associated with which value of the program counter.

The results indicate that the range table approach outperforms the setjmp implementation by a factor 100, while it is approximately 3.5 times as fast as the nested function design. For this reason we think the machine dependent solution to be the most promising on the long term. At this moment, however, the nested function implementation is also very attractive, because of the portability of its code. In all implementations, problems arise when we decide to compile the code with optimisation options switched on, so a solution for the optimisation problem is sketched. We think we have succeeded in extending C with very usable exception handling facilities. We will call our exceptions to C translating compiler “XTC”.
Acknowledgements

The implementation of exception handling facilities in C was conducted as part of the Pegasus project, a joint research project of the University of Twente and the University of Cambridge, aimed at the design of an operating systems architecture for scalable distributed multimedia systems. Part of my work was carried out at the University of Twente, but a considerable amount of the research was also conducted at the Computer Laboratory of Cambridge University. The successful completion of the project would have been impossible without the help of quite a number of people.

First of all, I would like to thank my supervisors in Twente, Sape Mullender (not only for his supervision in general, but also for giving me the opportunity to work in Cambridge), Martijn van der Valk (for all the time he spent reading my preliminary reports) and Richard Earnshaw (for all the important contributions and suggestions he made and all the time he spent looking for bugs in my programs and fixing them). Richard’s suggestions for the optimisation problem, especially, were of great value for the range table implementation.

Then I would like to thank David Evers of the Cambridge Computer Laboratory, for all his help on the range table compiler. I am also indebted to all the other people in the Lab, for making me feel at home and making my time in Cambridge a very pleasant one. I would especially like to mention the rest of the Sleepy Hollow population: Robin Fairbairns, Mian Wei, James Hall and Timothy Roscoe (for his suggestions concerning my haircut). My gratitude also goes to Ian Leslie and Derek McAuley, because they enabled me to work at Cambridge University and arranged a room for me in Fitzwilliam College.

Next, I would like to say thank you to all the graduate students in the “Systems Software and Architecture” group of the department of Computer Science of the University of Twente. I have had a very enjoyable time working there, and especially valued the numerous coffee breaks.

Finally, I should not forget to gratefully mention Marieke Toes, for all her support, but more importantly, because she insisted on it.
## Contents

1 Introduction .................................................. 1

2 Exceptions .................................................... 3
   2.1 Introduction ............................................. 3
   2.2 Exceptional conditions ................................. 3
      2.2.1 Definition and nature of exceptions ............ 3
      2.2.2 Operation failure ................................ 5
   2.3 Use of exceptions: raising and handling ............. 6
      2.3.1 Raising and handling ............................... 6
      2.3.2 Issues ............................................. 7
      2.3.3 Try blocks ........................................ 9
   2.4 Default and finally handlers .......................... 10
   2.5 Return codes ............................................ 12
   2.6 Exceptions are rare .................................... 13

3 Nemesis ..................................................... 15

4 Existing exception handling implementations ............ 17
   4.1 Introduction ............................................. 17
   4.2 Condition signalling in PL/I ............................ 17
   4.3 Exceptions in CLU ....................................... 18
   4.4 Signals and errors in Mesa ............................. 18
   4.5 Throwing and catching exceptions in C++ .......... 19
   4.6 Other implementations .................................. 20

5 Purpose of the project ..................................... 21
   5.1 Exceptions in C .......................................... 21
   5.2 Requirements ............................................ 22

6 Exceptional syntax ........................................... 23
   6.1 Introduction ............................................. 23
   6.2 Exceptional C: syntax example ........................ 23
   6.3 Declaration of raisable exceptions .................... 23
   6.4 Associating handlers with try blocks ................. 24
      6.4.1 Finally handlers ................................ 25
   6.5 Nesting .................................................. 26
7 Portable implementation of the XTC compiler

7.1 Introduction .................................................. 29
7.2 A simpler syntax ............................................. 29
  7.2.1 A different way of passing arguments in the first compiler .... 29
  7.2.2 Nesting .................................................. 30
7.3 Handling the exception ....................................... 30
7.4 Implementation: saving and restoring contexts .......... 31
  7.4.1 The first XTC compiler .................................. 31
  7.4.2 Extending an existing C compiler ..................... 32
  7.4.3 Recognising and checking exceptional C .................. 33
  7.4.4 Checking the input code: data structures ............... 34
  7.4.5 Handling exceptions: the code ......................... 38
  7.4.6 Core of the handling mechanism: setjmp/longjmp ......... 40
7.5 Portable exceptions using nested functions .......... 42
  7.5.1 Introduction ............................................ 42
  7.5.2 Nested functions and exception handling ............. 43
  7.5.3 Issues and problems ................................... 44
  7.5.4 Nested functions exception handling: implementation .. 45
  7.5.5 Summary ............................................... 46

8 XTC-2: low overhead exceptions .......................... 49

8.1 Introduction .................................................. 49
8.2 Syntax issues ................................................ 49
  8.2.1 Passing arguments ..................................... 50
  8.2.2 Explicit nesting ....................................... 50
8.3 Outline of the implementation ............................ 51
  8.3.1 Stack unwinding versus context saving .................. 51
  8.3.2 Range tables .......................................... 52
8.4 Implementation of the mechanism ......................... 54
  8.4.1 Building the try tree .................................. 54
  8.4.2 Generating range tables ............................... 56
  8.4.3 Translating exceptional C ............................. 57
  8.4.4 Implementing stack unwinding .......................... 58
  8.4.5 Range tables and output code in C .................... 59
  8.4.6 Summary ............................................... 60

9 Results and performance ......................................... 63

9.1 Results ...................................................... 63
9.2 Performance .................................................. 63
9.3 Optimisation ................................................ 69
  9.3.1 The optimisation problem .............................. 69
  9.3.2 Solving the optimisation problem ...................... 71
  9.3.3 General solution for the optimisation problem .......... 72
  9.3.4 Performance of optimised code ........................ 75
10 Conclusions
  10.1 Evaluation .............................................. 77
  10.2 Future work ............................................. 79
A A complex example ........................................ 83
B Ideal implementation in RTL .............................. 85
C Example range table program ......................... 89
# List of Figures

2.1 Language constructs .................................................. 10  
2.2 Scopes of nested try blocks ........................................... 11  

6.1 Proposed syntax .......................................................... 24  
6.2 Transfer of control ........................................................ 25  
6.3 Transfer of control: finally handlers .................................. 26  

7.1 Handling exceptions: example ........................................... 31  
7.2 Handling exceptions: executing the handler ......................... 32  
7.3 Contagious exceptions data structure ................................ 35  
7.4 Try tree data structure .................................................. 36  
7.5 Global exception tree structure ....................................... 37  
7.6 Chained list of context blocks ......................................... 39  
7.7 Exception handling: sketch of implementation ...................... 41  

8.1 Nested try statements .................................................... 51  
8.2 New try tree data structure ............................................. 52  
8.3 Disjunct ranges in program code ...................................... 53  
8.4 Unwinding the stack ...................................................... 54  
8.5 Fields of try and handler nodes ...................................... 56  
8.6 Translation of exceptional code ....................................... 58  
8.7 Stack and range table .................................................... 59  
8.8 Combination and generation of files into a program ............... 61  

9.1 Number of process cycles for nested functions and range tables 68  
9.2 Compiler optimisation: dead code .................................... 69
List of Tables

9.1 Performance (no optimisation) .................................................. 65
9.2 Performance (function call in try clause) .................................. 67
9.3 Performance of raise statements ................................................ 68
9.4 Performance (with optimisation) ................................................ 76
Chapter 1

Introduction

As present day computer systems are becoming increasingly complex, it can be observed that their usefulness, the scope and quality of services provided, strongly depends on the development of reliable software. The software should be able to efficiently deal with the system's complexity in a structured way. The intricate interdependencies between different programs that are running at the same time and even between distinct modules of a single large program, make the task of keeping track of all possible events that may take place, and of dealing with them appropriately, increasingly hard. The sheer size of the computer systems under consideration virtually prevents the programmer to maintain full knowledge of the exact behaviour of each program module used in these systems. Their becoming distributed only helps to exacerbate this problem. There are many places where something may go wrong and many ways in which a program can crash or run into trouble caused by errors.

Suppose, for example, that a client sends read requests to a file server, which processes the requests and sends back the reply. Certain circumstances, however, may prohibit the file server to properly carry out the request and to provide the client with the requested data. The cause of such a problem may be that the file is unreadable, or a disk has crashed or the file server is overloaded. In any case, the client should be notified of the event, so that it can react with the appropriate actions. One way of doing this is to let the file server return a special value, indicating what has happened, e.g. by way of error codes. It is the responsibility of the client not to interpret this return code as the requested data, but to recognise it as an error indication and to act accordingly. Now, problems arise, if the programmers at the client side are not completely aware of exactly what special codes may be returned by the file server or if they forget to examine the reply closely, and therefore interpret the returned code as the requested data, resulting in unspecified behaviour of the program.

The development of reliable software, however, requires that each program module must be defined to behave reasonably under a wide variety of circumstances [Liskov et al., 1979]. Its specifications should provide a well-defined response to all possible combinations of legal inputs (inputs satisfying the type constraints), even when lower level procedures on which this module depends fail and even in the face of other errors. Failures of procedures and errors in general are examples of so called exceptional conditions. One of the essential problems in writing reliable software is to respond to such exceptional conditions.

Another way of dealing with exceptional conditions is to provide explicit exception handling facilities in the programming language. It will be shown that exception handling mechanisms provide an elegant way to deal with exceptional situations: they enforce a control
structure that cannot be ignored by forgetful programmers [Evers, 1994]. A well-constrained and structured notation with regard to exceptions and exception handling was first introduced by Goodenough as a means to deal with these matters [Goodenough, 1975] and they have later been implemented in a number of different languages as a convenient way of doing so. In the field of operating system design, however, where such a mechanism would be extremely useful, the dominant programming language is C, which does not provide exception handling facilities at all.

In this report we will describe the progress made in extending C with exception handling as an integral part of the programming language (that is, with all the syntax checking and type checking required). We will call the resulting compiler the XTC compiler ("eXceptions To C" or "eXceptions Translating C" compiler). It will be shown that there are really two distinct approaches to solve this problem. The first approach attempts to arrive at a portable (machine independent) implementation of the exception handling mechanism, either using the relatively inefficient *setjmp* and *longjmp* routines, or using the GNU C extension of nested functions. The second approach attempts to do without the *setjmp* routine, resulting in a very efficient program code (almost without overhead), for those cases in which no exception is raised (normal cases), at the cost of portability. It uses some machine specific instructions to find the appropriate handler for dealing with the exception.

There is a lot of ambiguity in the terminology with respect to exceptional situations and exception handling. Chapter 2 provides us with an acceptable and workable definition of the relevant terms and addresses the nature of exceptions and the way to deal with them. The research was conducted as part of the Pegasus project and in particular of that part of the project that implements the kernel: Nemesis. Chapter 3 will therefore briefly describe this project, giving us a bit more insight in the context of the research. The next chapter discusses some existing implementations of exception handling facilities with an eye to evaluating some design decisions in these languages that are also of relevance to the XTC compiler. Before we start our research, we should first agree upon the goals and the purpose of the project. Chapter 5 is devoted to dealing with these matters and it provides a list of requirements that the eventual exception handling mechanism should satisfy.

Another issue that needs careful consideration is the syntax that is to be used for the exception handling mechanism. Chapter 6 introduces such a syntax for the input language "exceptional C", that is both simple and powerful.

After that, it is time to start implementing. It was mentioned before that there are roughly two approaches to tackling the problem: portable and non-portable. For the former, we have used the machine independent *setjmp* routine (as suggested by [Roberts, 1989]), which will be discussed in Chapter 7. In this chapter a complete compiler is built for the proposed syntax and here we will also mention an alternative portable implementation that uses nested functions, instead of *setjmp*. For the latter approach, however, no compiler has been built, because of insufficient time.

Since we were not entirely satisfied with the results, we then started working on a second (non-portable) version of the XTC compiler. This version removes the rather inefficient *setjmp* instruction and implements a genuine *stack unwinding* mechanism. It will be discussed in Chapter 8.

Chapter 9 discusses the results with respect to the overhead of the different approaches and also briefly reviews the problem of optimisation. Chapter 10, finally, contains an evaluation of the mechanism that we have implemented and also provides a number of suggestions for future work.

---

**Exceptional C: design and implementation of the XTC compiler**
Chapter 2

Exceptions

2.1 Introduction

In the development of complex software, great care must be taken that all possible events that may occur are appropriately dealt with, in order to make the program reliable in a wide range of circumstances. In particular, it should be able to cope with all the events that are no part of the normal program behaviour, but which could lead to erroneous results, should an appropriate handler for them be omitted. However infrequent they may occur, the program ought to provide a means to deal with them nonetheless.

Suppose, for example, that a program requests a chunk of memory to perform a number of computations, say the sorting of a list of names. In this case the main task is to execute an efficient sorting algorithm. In fact, a program implementing exactly that may work fine for a long period of time, until one day, for some reason the system fails to allocate the requested memory. If the software is not able to cope with this situation in a proper fashion, that is, if this allocation failure is unaccounted for, the program is likely to crash. This may be acceptable in some cases, such as certain user programs, but on other occasions it is most certainly not. If nothing else, it is not very pleasant.

This is especially the case in the design of operating systems, where it is crucial that the software behaves according to its specifications, even in the face of errors or malfunctioning of one of its components.

Errors, however, are but one of the possible conditions that should be dealt with. In this chapter, we will have a closer look at what exceptions are and what they can be used for. The description will be very theoretical and general in the sense that, where possible, we will not commit ourselves to a specific syntax or implementation. Only key concepts are introduced, the actual language constructs will be dealt with in later chapters. In the last section, we will briefly discuss the possible alternative to exception handling and the advantages and problems associated with them.

2.2 Exceptional conditions

2.2.1 Definition and nature of exceptions

Although the nature of exceptional conditions has been touched upon intuitively already, no formal definition has been provided so far. We will find that there are many causes for ex-
ceptional conditions, some of which may not be so obvious. For a thorough understanding of exceptions and the use of exception handling facilities it is necessary to look at such conditions in a little more detail. From now on we will assume the words “exceptional condition” and “exception” to be interchangeable and both will be used in this thesis.

As has been observed in Chapter 1, writing reliable software implies that certain requirements should be met, paramount among which is that the program should behave reasonably in a wide variety of situations, even when failures arise in the lower level software or the hardware, or, in general, when events take place that can be classified as being abnormal [Ghezzi et al., 1990]. In practice, however, such a requirement is not very useful, because it suffers from the obscuring fact that it is not entirely clear what is meant by normal and abnormal events (nor does it specify how reasonable behaviour should be interpreted). This depends solely on the design decisions taken by the programmer and will vary from application to application.

For this reason, we should refrain from merely defining exceptional conditions to be abnormal conditions of rare occurrence. Although true, such a definition would not be a very practical one. A slightly better definition, in the same line, is given in [Szalas, 1985]: “in general, exceptions are rare, often faulty situations detected by a run-time system or by a user program.” Although better in the sense that it takes into consideration the detection of the situation, it falls short of indicating what the nature of the situation might be. Instead, we will use the more practical definition for the concept of exception that was suggested by Turba:

\textbf{exception} “an exception is a state of execution that occurs when an operation is invoked and the action of the operation cannot be completed properly” [Turba, 1985]

In addition, we assume exceptions to be either detected by the user program or by the run-time system. Observe, that in the above definition no implication is made whatsoever as to whether such a state of execution is caused by errors or not. In fact, we agree with the point of view taken by [Goodenough, 1975], where exceptions are considered to be the result of a number of possible causes, only one of which comprises errors and failures (albeit probably the most important one).

The detection of exceptional conditions by the run-time system or user program can be specified a bit more accurately by defining that of all conditions detected while attempting to perform some operation, \textit{exception conditions} are those which are 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 the invoker’s response is called \textit{handling} the exception [Goodenough, 1975].

\section*{Exception properties and operation generalisation}

Some basic properties of exception conditions relevant to language features for dealing with them are the following [Goodenough, 1975]:

\begin{itemize}
  \item an exception’s full significance is known only \textit{outside} the invoked operation;
  \item the invoker may be permitted to terminate the operation at the point of the exception’s detection.
\end{itemize}
This means that the responsibility of handling the exception lies at the invoker's side. Thus, upon detection of an exception condition, the operation is stopped and control is transferred to the invoker.

We furthermore define:

operation's domain as the set of inputs for which outputs (effects) have been defined;

operation's range as the effects that are obtained when certain inputs are processed.

With these definitions we can further specify the nature of exceptions. In essence, exceptions permit the user of an operation to either extend the operation's domain or its range, by specifying its behaviour in the event of inputs that were originally not part of the domain. In other words, exceptions serve to generalize operations, making them usable in a wider variety of contexts than would otherwise be the case.

2.2.2 Operation failure

In his paper Goodenough distinguishes three categories in which the use of exceptions may fall. They will be looked at briefly in this paragraph, after which we will focus primarily on the nature of exceptions caused by failures. The three classes of usage identified by Goodenough [1975] are the following:

- permit dealing with an operation's impending or actual failure;
- indicate the significance of a valid result or the circumstances under which it was obtained: result classification;
- permit an invoker to monitor an operation, e.g. to measure computational progress or to provide additional information and guidance should certain conditions arise.

What these three uses have in common, is that they are all concerned with exceptional conditions in the true sense of the word. Errors, clearly, can be expected to occur infrequently and prohibit proper execution of the operation. Result classification indicates how a certain obtained value should be interpreted. Suppose, for example, a procedure is reading a list of items. The last item in the list has an indication saying this is in fact the last item. Reading this item, produces a correct result, which should be interpreted as such and should also prevent the procedure from attempting to read even more items.

With respect to the third way of using exceptions, it should be remarked that, although this clearly has to do with exception conditions, it generally requires an exception handling mechanism that is able to continue the operation that produced the exception situation. This means that control has to be returned to the operation, yielding less structured code.

Even though the resulting mechanism is very flexible, this very same mechanism proves to be rather difficult to use for inexperienced programmers and, worse still, facilitates the unsafe programming fashion of removing symptoms of an error instead of the cause of the problem [Ghezzi et al., 1990]. For example, it may deal with the exception of an unacceptable operand value by just generating a random value that will be accepted. For this and other reasons, that will be addressed later, we will refrain from implementing such a resume mechanism ourselves.
In this paragraph we do not intend to discuss the use of exceptions at great length, but we do want to examine the failures of the first category more closely. They clearly constitute exceptional conditions. Moreover, they are probably the most important class of exceptions and therefore deserve some further attention.

There are two types of failures, that are of relevance to language features in connection with exceptions:

- range failures
- domain failures

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 correct result), or decides it may not ever be able to satisfy its output assertion. Suppose, for example, that a read operation encounters a parity error when attempting to read a record. The input itself is correct in the sense that the record can actually be read, the problem is that the operation produces a result which it knows to be wrong. Another example of a range failure is when a numerical program finds evidence of divergence of the algorithm.

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

2.3 Use of exceptions: raising and handling

It should be clear by now what exceptions are, but we have said hardly anything about the use and usefulness of exception handling. Exception situations may arise during the execution of a function, or a module, or even an operation itself, but the interesting question is now, what will be done with them. To answer this question we will first introduce a number of key concepts.

2.3.1 Raising and handling

Upon detection of an exceptional condition at runtime, a procedure needs some way of communicating this fact to its caller. The operation of doing so in an exception handling mechanism is called raising the exception. Raising exceptions can be done explicitly, by executing a raise command in the procedure, but in many implementations (e.g. Python, CLU), they can also be raised implicitly. Implicit raising means that in certain situations (for example, when an attempt is made to divide by zero) the exception is raised automatically by the implementation of the programming language itself.

On the other side of the exception handling mechanism there should exist a piece code that can deal with the exception. Dealing with an exception is called handling the exception and the corresponding code is called the handler. Now, when a procedure detects an exceptional condition and raises the appropriate exception, this should cause the corresponding handler to be activated, i.e. the control should be transferred to the exception handler (in paragraph
6.4 and more particular in figure 6.2 this transfer of control will be illustrated for the syntax we propose to use for exceptional C).

With “corresponding handler” we mean the handler that matches the exception. In general a match is established if the handler is explicitly defined to deal with exceptions of this particular name. This, however, is not always the case, as will be shown in paragraph 2.4. In any case, the exception handler must reside somewhere outside the operation that raised the exception.

2.3.2 Issues

Although the scheme described above appears to be very straightforward there are a number of non-trivial issues that still need to be resolved. In general, the most important issues can be summarized as follows:

1. How is an exception declared and what is its scope?
2. How is an exception raised?
   - are explicit raises possible?
   - are implicit raises possible?
3. How do we define exception handlers?
4. How are raised exceptions associated with handlers?
5. Where will the program execution resume after the exception is handled?
6. Is it possible to pass arguments to the exception handler and, if so, what kind of arguments?

Declaration, scope and association

Issues 1, 3 and 4 are closely related and need careful consideration. With respect to handler definition and exception scope, one can imagine an exception mechanism in which exceptions and handlers are declared at the top level only, more or less like functions in C. This means that there is exactly one handler for each exception and this handler deals with that particular exception, regardless of where it is raised. The advantage of such a scheme is that it would be relatively easy to implement. It is, however, extremely rigid.

Alternatively, one could think of a mechanism where the definition of handlers, and the scope of exceptions is completely hierarchical. In this way the scope of an exception is limited to the code that is “underneath it” in the hierarchy. Furthermore, it means that a redeclaration of an exception by redefining the exception handler at a lower level in the hierarchy, creates a new scope for this exception, which from that point supersedes the previous one. Scopes of exceptions will be addressed more thoroughly in paragraph 2.3.3.

The issue of associating handlers with exceptions is dealt with quite elaborately by Goodenough. In brief, he identifies three types of errors that programmers are likely to make, unless the language provides some way of preventing them [Goodenough, 1975]:

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 (i.e. the place from which an operation is invoked).

3. Associating a handler with the wrong exception.

The proper way of dealing with these error possibilities is to devise a notation that makes compile time detection of errors possible. This would prevent programs to crash or display unexpected behaviour at run time. For this purpose, it is reasonable to require:

- explicit declaration of what exceptions an operation can raise, so that the compiler can check whether handlers exist for each exception that can be raised,
- static association of exception handlers with activation points (i.e. associations defined at compile time rather than at run time), to prevent problems (2) and (3), but also to make the program faster: overhead introduced by the association will no longer occur at run time.

To satisfy the requirement for static association, handlers are attached to syntactic units containing activation points. In other words, an exception handler is statically bound to a region of text by way of a new language construct [Turba, 1985]. These syntactical units, which we will call try blocks are discussed in paragraph 2.3.3

Raising exceptions

When designing exception-handling facilities to enable programmers to write reliable software, the whole objective is to support, at least, the explicit raising of exceptions. Whether you want the underlying systems to activate exceptions implicitly is of no real importance at this point. It could easily be done at a later stage, since it only implies replacing all the current run-time errors with raising of exceptions, for which built-in handlers must be provided.

Closely related to this is the issue of user defined exception handlers. Nearly all proposals on exception handling suggest that it should be possible to allow programmers to define their own exception handlers. We will adopt this view without reluctance, considering it a basic necessity to make the handling mechanism useful. After all, from the very outset the goal was to provide programmers with the tools to deal with exceptional conditions in a way they specify to be right.

Resume or abort

Concerning the resumption of execution after the handling of an exception, we have already shown that, for several reasons, it may not be such a good idea to permit the operation that raised the exception to resume its work. In our opinion, the flexibility gained by such a facility is outweighed by the disadvantages observed in paragraph 2.2.2., in addition to the problem of its need to store much more state in order to let the action be resumed in the exact same context as on the moment when it was suspended. For these reasons many implementations (e.g. Bliss, CLU, Ada) do not support the resumption operation [Liskov et al., 1981]. Instead, the operation is simply aborted and the operation's activation record is destroyed. Still, there are a few languages that do provide such facilities (e.g. PL/I and Mesa) and it is therefore useful to bear in mind that there are roughly two models with respect to this issue (Mesa actually provides a few refinements to these two models, which we will ignore here for the sake of brevity):

---

**Exceptional C: design and implementation of the XTC compiler**
• resumption model
• termination model

These two models are not disjoint. In Mesa, for example, a programmer can specify operations to be aborted, resumed or even skipped. Resuming an operation after handling an exception closely resembles executing a function call and returning. In fact, for most applications, we can use functions to deal with this type of handling. So, even if we do not consider all the disadvantages of exception handling according to the resumption model, there is still no great need for facilities that support it.

Argument passing

The last issue deals with the possibility of passing arguments to an exception handler. Many implementations do not support this feature at all, or only in a very limited way [Liskov et al., 1981], [Turba, 1985], [Ribbers et al., 1989]. We feel that an exception handling mechanism without argument passing to handlers has lost much, if not all, of its usefulness, because, clearly, sometimes, additional information is needed to deal with an exception condition properly. Furthermore, we suggest to allow the types of the arguments that may be passed to be as varied as possible.

2.3.3 Try blocks

Since it may be advantageous to have distinct handlers for the same exception for operations that would otherwise be at the same level of abstraction in a program (e.g. a sequence of statements of a function in C), we want to encapsulate the code that may raise a specific exception and associate handlers with the encapsulated block of code, rather than with all the code at that level. Another reason for doing this, is that most (or some) program code may not raise exceptions at all, so we do not want to burden that code with potential overhead, however small, caused by exception handling. Finally, to satisfy the requirement for static association, we need to attach handlers to syntactic units as observed in paragraph 2.3.2.

For this purpose, we introduce the concept of a try block. A try block contains the code that may raise exceptions. Exception handlers will somehow be linked to try blocks, so that when an operation in the try block raises an exception, it will be handled by the appropriate handler associated with the try block. If no such handler exists, we will look for a handler that is associated with the try block found in the next level up in the hierarchy, and so on, until eventually a handler is found or the top level is reached. The exception handling facilities are then confined to the try blocks. The reach or scope of a handler is the syntactic unit (the try block) to which it is attached, including all the contained syntactical units that do not have a handler for the same exception themselves.

So far, we have discussed the following three concepts that have to be represented by a language construct, regardless of what the syntax of the language looks like and regardless of how the mechanism is implemented (this is also illustrated in figure 2.1):

• try block construct
• raise construct
• handler construct
Figure 2.1: Necessary language construct for exception handling

Instead of saying that a try block encapsulates a piece of code, a try block could also be looked upon as being a kind of protected area that is guarded by exception handlers. To make the structure hierarchical, somehow, nesting of try blocks should be possible. There are really two ways of doing this:

1. we could allow the language construct for a try block to be explicitly nested (more or less like a compound statement in C),

2. or we could have try blocks which are intrinsically flat and support nesting only by way of doing function calls inside a try block, where the called functions have try blocks of their own. This corresponds to a certain extent to the notion of functions in C, where functions may not be defined in a nested fashion, but where nested function calls are allowed.

Note that both methods of nesting are equally powerful in the sense that they allow the construction of the same hierarchy of exceptions. The only difference is that the latter may require a bit more work for doing so (namely the definition of the functions). Ideally, both ways of nesting should be supported. The issue of the scope of exceptions and exception handlers is illustrated in figure 2.2. In this figure we assume there is only one exception. The small triangles indicate the start of a try block underneath it, whereas the shading of the large triangles shows the scope of the exception handlers. Note that an exception handler itself is also part of the scope of the handler above it. This means that an exception raised here will be handled by a handler one level up.

2.4 Default and finally handlers

So far, our discussion has concentrated on handlers that deal with one exception only, namely the exception for which it was defined and which should match the exception raised. It is
very convenient, however, to define default handlers. Default handlers match any exception. If such a handler is specified, it will be executed whenever an exception is raised, unless it is explicitly overridden by the definition of a normal handler.

Default handlers appear to be very useful, for example to prohibit the occurrence of unhandled exceptions. All we have to do is to provide a default handler at the top level and no matter which exception is raised, we are sure that it will be dealt with. If such a default handler at the top level actually catches an exception, it may indicate that something is wrong (otherwise there would have been a normal handler for the exception), but at least we are now able to exit gracefully.

Even when nothing is wrong, default handlers are said to be useful [Goodenough, 1975] to programmers. In that case, we use the default handler as a general handler for a whole set of exceptions. Exceptions having such default handlers are said to be default exceptions. Failure to provide a handler for a default exception is not an error. On the contrary, it is a way of specifying that the default handler is to be executed. In other respects default exceptions are the same as exceptions that do have a handler defined for them.

Default exceptions make detecting errors in the use of exceptions more difficult, because, if there exists a default handler at the top level, the compiler can no longer detect whether a programmer has forgotten to supply a handler for a certain exception, since there is always the default handler that can handle it. For this reason Goodenough proposes exceptions with default handlers to be declared as such beforehand.

In our opinion such a declaration makes the mechanism needlessly complicated. Moreover, because of the observed problems with default exceptions, I think we should refrain from using default handlers unless it cannot be avoided.

There are only a few cases I can think of in which default handlers are really needed. One of them is a distributed system, in which a program, for example a client, is not sure what exceptions may be raised by its environment (e.g. by the servers it accesses). Even here, it would be far better if the client did know all the exceptions it may expect and provided an appropriate handler for them. This, however, may not always be possible, since new

Figure 2.2: Scopes of exception handlers in nested try blocks
exceptions may be defined and raised, of which the client has no knowledge beforehand.

Another case, where default handlers are very useful, if not indispensable, has to do with libraries. The author of a library can detect run-time errors but does not in general have any idea what to do about them. The user of a library may know how to cope with such errors, but cannot detect them, or else they would have been handled in the user's code and not left for the library to find [Stroustrup, 1993]. The notion of an exception helps to deal with these problems. The idea is that the library functions raise the exceptions, while the user code provides the appropriate handlers for them. This allows for dealing with erroneous conditions in library functions in a uniform and structured manner.

One library however, may raise a great number of exceptions, one or several for each function it contains, and it would be a very tedious job for the programmer, to provide a handler for each one of these, even when he is sure some of them will never be raised. Moreover, consider a set of user programs that is supposed to reliably work with a specific library (meaning that all exceptions are accounted for). Suppose, furthermore, that at a certain point in time the library is extended with a new exception (which may very well happen). It would be quite painful now, to have to modify all of these user programs, so that they will also handle the new exception appropriately. In these cases, default handlers can help out.

In all other cases we are inclined to consider the use of default exceptions (however convenient) as bad programming. If you want to use an exception of a certain name, you should also be willing to provide a handler for it.

Finally handlers

A special case of default handler is the finally. Despite the objections against normal default handlers, finally's are extremely useful. They are not normal defaults. They allow programmers to cope with a problem that is, at least partly, caused by the very use of exceptions. Suppose, for example, that in an environment in which resources are shared, an operation inside a try block has succeeded in acquiring a certain number of mutexes (mutual exclusion semaphores). If, at that moment, an exception is raised, control is transferred to the handler, without releasing the mutexes. The handler, however, does not know where exactly in its scope the exception was raised (which could be anywhere), let alone whether the operation has succeeded in claiming resources, and if so, which ones and so on.

For this reason we introduce the finally handler. A finally handler will always be executed regardless of the fact whether an exception was raised or not. If an exception was raised when the finally clause was entered, it will be re-raised after the finally code has been executed. This suits the observed problem perfectly: in the finally handler we can establish all lock-releases of the mutexes that have been acquired up to now. If no exception was raised, the code will just continue with the instruction after the finally handler and otherwise a re-raise instruction will be executed (this flow of control will be illustrated for the syntax we propose to use for exceptional C in paragraph 6.4 and more particular in figure 6.3).

2.5 Return codes

The traditional way of dealing with exceptional conditions communicates the existence of such a condition to the operation's invoker by means of return codes. On encountering an exceptional condition the function or module called will simply return a special value ("funny
code"), relying on the invoker to check the results carefully and deal with it accordingly. We will now demonstrate why this scheme is dangerous and why exception handling is so useful:

- return codes can be ignored by forgetful programmers, whereas exceptions cannot: an unhandled exception can be detected at compile time, or alternatively cause the program to halt;
- it isn’t always feasible to let the return code represent the exceptional condition (unless at great expense), because sometimes there are no “funny codes” available; consider, for example, a function that may return any integer value; in that case there is no value we can use for exceptions;
- return codes have to be checked at each level of abstraction; in exception handling the control scope is automatically unwound until the appropriate handler is found ("funny codes" are no longer needed);
- because return codes have to be checked no matter what, they are likely to constitute a considerable overhead during program execution; if carefully implemented, exceptions need not incur runtime overhead in normal cases at all;
- exceptions clarify both the control structures and the specifications of programs, by clearly separating the normal and abnormal cases.

We therefore conclude that exception handling facilities constitute a versatile tool to help write reliable software. They enforce a control structure that cannot be ignored by forgetful programmers.

### 2.6 Exceptions are rare

Although there is no general agreement on this point, we will assume exceptions to be rare. With this point of view we oppose the thoughts expressed in [Goodenough, 1975], where it is felt that exceptions “are not necessarily rarely activated.” The very nature of the word “exceptional condition”, however, indicates that these conditions are not likely to occur frequently. Most implementations share this opinion ([Turba, 1985], [Szalas, 1985], [Ribbers et al., 1989]).

As a consequence, an implementation should attempt to incur no runtime overhead or as little as possible in the normal case (i.e. the situations in which no exception is raised), whereas the handling of an exception may be relatively slow. Since this is not likely to happen very often, it doesn’t really matter whether the handling is done very efficiently or not. Implementing a no overhead implementation in the normal case, has the additional advantage that it will not discourage programmers to use it, thus making their programs more reliable (at least they are not punished by means of considerable overhead when doing so).
Chapter 3

Nemesis

The exception handling mechanism discussed in this thesis is being developed under the umbrella of the Pegasus project. Pegasus is a project aimed at the design of an operating systems architecture for scalable distributed multimedia systems and the development of a working prototype as the demonstration of the architecture [Mullender, 1992]. The project defines as a central part of its work, a kernel, tentatively called "Nemesis" which will act as a basis for the operating system software that the project will produce [Fairbairns, 1993].

Nemesis will be used to support systems typically distributed over a small number of workstations. It is anticipated that the base functionality required to support these systems will also be applicable to the construction of a range of other systems. Within a group, the workstations are assumed to be connected via a high speed multi-service network, which is capable of delivering to the workstations real-time audio and video data in addition to the usual LAN traffic such as file system data.

Nemesis is designed to take advantage of the benefits offered by a single virtual address space and a Nemesis system consists of a number of distinct schedulable entities called *domains* which interact in the manner of *clients* and *servers*. A domain comprises an address space and some number of threads, the management of which is performed by a user-level thread scheduler.

Nemesis code and interfaces are expected to be written in a style which makes use of exception handling to deal with all exceptional situations. To facilitate this process, exceptions are declared in Middl interfaces (Middl is the Nemesis Interface Definition Language [Roscoe, 1994]). In an operating system and particularly a distributed one, a great many resources are managed and a lot of interdependencies between various components can be identified. Therefore, a reliable and efficient implementation of exception handling is required.

Furthermore, since Nemesis code is written in C, there is need for such an implementation in the C programming language. As a first attempt in that direction a portable set of C++ macros has been implemented in the DEC Research Center [Roberts, 1989]). A similar approach is proposed for Nemesis, with an eye to a slightly more efficient implementation.

Having said all that, it should be remarked that the implementation of the exceptions in C is not at all restricted to the Nemesis kernel. On the contrary, it should be generally applicable in any C program and, as such, stands on its own.
Chapter 4

Existing exception handling implementations

4.1 Introduction

Exception handling facilities have already been implemented in a considerable number of programming languages. Some of these implementations provide only very limited facilities (e.g. without argument passing, hierarchical scopes or reraises), while other languages (such as PL/I and Mesa) support extremely complicated raise and resume mechanisms. It is beyond the scope of this document to elaborate all the different implementations. To get an idea of how exception handling is used in practice and to place the design under consideration in its proper context, however, it is usefull to mention just a few of them briefly, together with some of their peculiarities.

4.2 Condition signalling in PL/I

In PL/I (Programming Language I, one of the first languages to provide explicit exception handling facilities), exceptions are called conditions. The language provides a number of built-in exceptions with corresponding handlers (e.g. ZERODIVIDE to deal with an attempt to divide by zero), but allows programmers to define their own exceptions as well. Moreover, users are permitted to overrule system handlers by redefining them. Exceptions are declared in ON statements:

ON CONDITION (exception_name) exception_handler

where exception_handler can be a single instruction or a block of code. An exception can be explicitly signalled using the instruction

SIGNAL CONDITION (exception_name)

If an ON statement is encountered at runtime, a new binding is created between the exception and the handler. The association of handlers with activation points is therefore completely dynamic. SIGNAL-ling an exception will result in execution of the handler that is currently associated with the exception. The handler is executed as if it was explicitly called as a
subroutine. This implies that, after handling the exception, execution is resumed at the point where the signalling was done, unless this operation is explicitly aborted in the handler.

The dynamic nature of binding and signalling, as mentioned above, leads to program code which is both difficult to write and confusing to read. It gets even worse by the impossibility to pass arguments to the exception handler, which means that the programmer has to resort to using global variables. According to some, all this renders the PL/I exception handling mechanism to be useless for most applications [Ghezzi et al., 1990].

4.3 Exceptions in CLU

The exception-handling facilities in CLU are in some respects less powerful than those in PL/I, but they are much easier to use. Furthermore, the CLU implementation allows the programmer to pass arguments to the handlers.

In CLU, exceptions can only be detected by procedures. When an operation inside a procedure encounters an exceptional condition, the procedure will return with that exception. The exceptions that may be raised by a certain procedure are declared explicitly in its header. CLU provides both user-defined and system exceptions. One of the main differences between CLU and PL/I is that CLU does not allow the resumption of an procedure that raised the exception [Ghezzi et al., 1990].

Another important feature is the static association of handlers with invocation points [Liskov et al., 1979]. This association of handlers with operations is done by way of an except keyword:

<operation> except <handler list> end

When the execution of <operation> raises an exception, the control is transferred to the <handler list>, which has the following form:

    when <exceptionlist i> : <operation i>
    ...
    ...
    when <exceptionlist n> : <operation n>

If the raised exception is contained in <exceptionlist i>, the corresponding <operation i> (the handler) is executed. If no handler list matches the exception the process is repeated at a higher level, and so on until a handler has been found or until the system handler failure indicates that there exists no such handler. When the handling is done, the program will resume with the statement immediately following the handler.

4.4 Signals and errors in Mesa

A very extensive implementation of exception handling can be found in the Mesa language, developed by Xerox Development Environment ([Mesa, 1984]). It offers a whole range of new statements designed to allow programmers to deal with exceptions in any way they think appropriate. Exceptions are called either ERRORS or SIGNALS. Upon catching a raised signal,
a handler (or catch phrase) is permitted, by way of explicit language constructs, to do such things as:

- **REJECT** the signal (propagate to the next visible catch phrase),
- **RETRY** the operation that SIGNAL-led the exception,
- **CONTINUE**, which leads to execution of the operation following the SIGNAL-ling operation,
- **RESUME** the operation, in which case the handler works exactly like a subroutine (and as such can even return results).

Although the Mesa exception handling mechanism is probably the most flexible scheme imaginable, the whole implementation is rather overloaded, which is not only a burden on the performance, but yields extremely complicated program code as well. In fact, in order to keep the performance acceptable, Mesa has never known a portable implementation.

### 4.5 Throwing and catching exceptions in C++

Because C and C++ are so closely related, and many of the issues at stake in the exception handling mechanism of C++ also apply to the implementation in C, we discuss the C++ handling facilities at a little more detail. First, however, it should be remarked that, although the syntax has been standardized, there are at this moment only very few full grown implementations available (if any at all [Stroustrup, 1993]).

Raising an exception is called throwing in C++ and an exception handler is said to catch raised exceptions. The exception handling mechanism is designed solely for dealing with errors, but this does not prohibit programmers to use it for other purposes as well. The notion of try blocks (see paragraph 2.3.3) is explicitly provided in the language by way of the keyword try. The way in which exception handlers are associated with try blocks and in which exceptions are raised is demonstrated in the following example:

```c++
try {
    if (condition) throw exception ;
}
catch ( exception ) {
    // handler code
}
```

The catch code can only be used immediately after a block prefixed with the keyword try or immediately after another exception handler. The handlers are (statically) bound to the preceding try block. It is possible to pass arguments to exception handlers in very much the same way as when passing parameters to functions.

The process of throwing and catching an exception involves searching the call chain for a handler from the throw point up through its callers [Stroustrup, 1993]. In this process the stack is unwound to the stack frame of the catching function and all local objects in the intermediate stackframes are destroyed by way of destructors. This implies that C++ follows the termination model.
Once a handler has caught an exception, that exception has been dealt with and other handlers that might exist for it become irrelevant. In other words, only the active handler most recently encountered by the thread of control will be invoked. This means that any exception thrown while executing a handler, must be dealt with by the caller of the try block. In addition, exceptions may be explicitly nested. So, for example, the following program is considered legal input:

```c
try {
...
}
catch ( exception ) {
  try {
  ...
  }
catch ( possibly other exception ) {
    ...
  }
}
```

Stroustrup, considers such nesting “rarely useful in human-written code and more often than not an indication of poor style” [Stroustrup, 1993].

A final remark concerns the acquisition of resources. The problem with respect to this issue is acknowledged, but there is no such thing as a finally handler. Instead, the proposed solution is to make careful use of the possibility to encapsulate access to resources in a C++ object. It is beyond the scope of this document to explain the scheme, and we will restrict ourselves to remarking that the mechanism works fine, but is certainly not as clear and straightforward as the use of a finally handler.

### 4.6 Other implementations

By no means should the selection of languages in the previous paragraphs be considered even remotely complete. There exist many other implementations and the languages described are only chosen, because they are quite distinct in the way they deal with exceptions and represent clearly the different schools of thought, or because they are closely related to our own future implementation (this applies mainly to CLU and C++).

Some other languages providing exception handling facilities are:

- Ada [Ichbiach et al., 1980],
- Modula-3 [Harbison, 1992],
- Pascal with exceptions [Turba, 1985],
- TUMULT OS Programming Language [Ribbers et al., 1989],
- LISP [Steele, 1990],
- Python [van Rossum, 1994].
Chapter 5

Purpose of the project

In the previous chapters the issue of the project’s purpose has already been touched upon briefly. This chapter attempts to define the goals and requirements more precisely. Careful consideration on the issues mentioned in chapter 2 is needed to result in a powerful yet usable implementation.

5.1 Exceptions in C

The previous chapter showed that there are quite a few existing languages providing exception handling facilities already. An obvious question would then be: why design a whole new exception scheme for C? To answer this question we should consider the present use of programming languages. In the field of operating system development, C is by far the most widely used language. C++, the language closest to C, has not been able to alter this fact. A great many very useful libraries and tools for implementing operating systems are available for this programming language. It is highly undesirable to have to rewrite all this code for a language like CLU, or make system programmers write their code in C++, while they generally find the C language much more suitable.

Furthermore, most of the existing system software that has to do with the Pegasus project in general, and the Nemesis kernel in particular, is written in C. From the very outset, one of the main objectives has been the provision of exception handling facilities to make the distributed multimedia systems architecture developed here more reliable. Therefore, they really should be implemented in C.

Another reason is that we are not quite satisfied with most implementations anyway. Many of them provide only limited support to the passing of arguments, and others, that do provide such facilities in an acceptable fashion are too complex to use and yield ill-structured program code (e.g. Mesa).

The only possible exception would be the mechanism of C++, but here the observed problem of C being the programming language in the field still holds. Many programmers think C to be by far the most suitable language for writing system code. Furthermore, even if one was to force every programmer to write his code in C++, we would still face the same problems, because there are hardly any compilers in which the exception handling facilities of this language are actually implemented, let alone in a decent manner. The very common GNU C++ compiler for example, recognises the syntax, but does not really do anything with it yet.
The arguments above, together with the observed usefulness of exception handling facilities, lead to the following definition of the project’s purpose: to extend the C programming language with language constructs that enable it to handle exceptions and to implement a compiler that can read this code, and translate it either to standard C or directly to executable code. We will call the extended C language “Exceptional C”, and the compiler “the XTC compiler” (“Xptions To C” or “Xptions Translating C” compiler).

5.2 Requirements

We still haven’t said anything about the requirements the mechanism will have to meet. The main demands our exception compiler should try to satisfy are listed below:

- explicit language constructs for the try block, raise function and exception handler,
- explicit declaration of exceptions that may be raised by a certain function,
- static association of exception handlers with try blocks,
- hierarchical definition and scope of exceptions in which a handler may re-raise an exception (this also implies that nesting should somehow be possible),
- handling based on the termination model (the operation that raises the exception will never be resumed),
- argument passing to exceptions in a straightforward manner and with as many different argument types as possible,
- exception handling mechanism should blend in nicely with the standard C programming language and the conventional C programming style,
- support for default and finally handlers,
- the runtime overhead in the case where no exception is raised should be as low as possible, while handling an exception may be relatively slow,
- the facilities are meant to deal with exceptional conditions in general, not just errors (although they are assumed to constitute the bulk of these conditions).

Most points have been dealt with already in the previous chapter, but the issue of the “exception handling mechanism blending in nicely with the existing programming language” is new. By this we mean that the very nature of the syntax and use of the facilities should be (at least to a certain extent) “C-like”. 
Chapter 6

Exceptional syntax

6.1 Introduction

In the previous chapter we have discussed the requirements that have to be satisfied by the compiler for exceptional C. Now we will propose a syntax that will enable us to effectively do so. The language constructs will be discussed, as well as the association and invocation of exception handlers. The implementation of the syntax by a real exception handling mechanism will be dealt with in Chapter 7 and Chapter 8. We should note, however, that the first implementation does not deal completely with the syntax proposed in this chapter. The syntax is strongly influenced by the exceptions in C++ and CLU and even more so by the notation proposed in [Roberts, 1989]. Nevertheless, there are a few features in exceptional C that can be found in none of the above.

6.2 Exceptional C: syntax example

The syntax of exceptional C tries to stay as close as possible to standard C with an eye on making the use of the facilities both easy and powerful. The proposed syntax can best be demonstrated by way of an example (see Figure 6.1). The example only shows certain aspects of the syntax, however. Things like nesting, default and finally handlers and so on are not shown. These issues, as well as some others, will now be dealt with separately.

6.3 Declaration of raisable exceptions

The first thing that strikes us, when we look at the example, is probably the explicit declaration of exceptions that may be raised by this function. Exceptions are declared by listing them (separated by commas) behind the keyword “raise”. For lack of a better word, we will call such exceptions contagious, because these are the conditions (possibly failures) that could not be cured locally and which are likely to infect the function at the next level up in the hierarchy. Strictly speaking they comprise the exceptions that may be raised inside the function, but for which the function does not provide a handler itself.

Note that the explicit declaration of contagious exceptions is not very common. Most existing languages do not support this feature at all. It is, however, extremely useful. It allows us to see at a glance, which exceptions may be coughed up by this particular function.
Figure 6.1: Proposed syntax for Exceptional C

Furthermore, it allows the compiler to check whether the exceptions are used correctly or not (i.e. whether handlers have been provided for each exception that may be raised), without descending all the way inside a tree of nested function calls. Suppose, for example, that function \( B \) is called within function \( A \) and function \( B \) declares contagious exception \( \xi \). In that case function \( A \) should either provide a handler for this exception, or be able to pass it on to yet a higher level. In other words, we can check the correct use of exceptions by just looking at the list of contagious exceptions of each function called.

Without the declaration, the compiler would have had to descend into function \( B \) (and if \( B \) in turn decides to call other functions, also into these functions, and so on, all the way until we hit bottom) to see if there are any exceptions that may be raised, for which there is no handler available. This amounts to an extensive search of the syntax tree, resulting in very long compilation times.

On the whole, we like to compare the declared contagious exceptions with the declared result type of a function. The type specifies what sort of result will be returned by the function (in the example of fig 6.1 “\texttt{void}”), the exception list specifies which exceptions may be raised by this function. Both indicate (potential) behaviour that escapes the contained function block.

### 6.4 Associating handlers with try blocks

Handlers are defined at the end of a try block. Their scope comprises all the contained try statements, beginning immediately after the \texttt{try} keyword and ending just before the start of first exception handler. Handlers can only be followed by other handlers or by the end of the try block. Figure 6.2 illustrates the flow of control both in the normal case and when an exception is raised.
6.4. ASSOCIATING HANDLERS WITH TRY BLOCKS

Special keywords will be defined for default handlers. The syntax for defining a normal default handler is:

```c
except default {
    /* default handler code */
}
```

So we see that no arguments can be passed to default handlers. The obvious reason for this is that default handlers may serve many different exceptions, caused by many different raise statements, each of which tries to parse its own set of arguments. This renders the task of providing an argument format for default handlers an impossible one.

6.4.1 Finally handlers

The last handler type, that was identified in Chapter 2, is the finally handler. To avoid confusion, we will prohibit the use of finally handlers and other handlers in the same try clause. If they would be allowed to be defined together, problems would arise in case an exception ε is raised for which there exists a handler in the current try block, while there is also a finally.

Because of its definition the finally handler would be executed first. When the finally has finished a reraise is done, but which handler is to be executed? The finally handler that raises the exception is no longer in the scope of the exception handler for ε in the current try block, so ε is passed to the next level up. This means that the exception handlers associated with this try block are never executed. To avoid these confusing situations, we will let the compiler generate an error when a user attempts to attach normal and finally handlers to the same try clause.

Finally handlers are defined by the following syntax:
finally {
    /* finally handler code */
}

There is also no argument passing to finally’s. There are two reasons for this:

1. even when no exception is raised (and therefore no arguments are specified), the handler is executed;

2. just like defaults the handler may serve many different exceptions.

To make sure the exception handling mechanism behaves as expected even in the face of raised exceptions that are first handled by a finally, we must therefore make sure that the resume at the end of such a handler is exactly the same as the original one (i.e. with the exact same arguments, etc.). All of this amounts to a slightly different flow of control in a program that uses the finally handler (see Figure 6.3).

Figure 6.3: Flow of control when an exception is handled in the presence of a finally handler

6.5 Nesting

To provide an hierarchical scope structure for exceptions (and exception handler), we need a way of nesting try blocks. There are two ways of doing this (see Chapter 2 are:

1. explicit nesting, also called lexical nesting: starting new try blocks inside other try blocks in the code (like compound statements),
2. **function call nesting:** nesting is only allowed by calling a function in try block and allowing this function to have its own *try block* in turn.

In our syntax, both types of nesting are allowed. When we start discussing the implementations, however, we will see that the first version of *XTC* only provides nesting at a function call level. This will be discussed more elaborately in paragraph 7.2.2. The second implementation also supports lexical nesting.
Chapter 7

Portable implementation of the XTC compiler

7.1 Introduction

Now that we have seen what exceptions are and what they are used for, as well as what the requirements for an exceptional C compiler are and what sort of syntax it is supposed to handle, it is time to start with the design and the implementation. In our first design, we try to arrive at a portable (i.e. machine independent) implementation of the exception handling mechanism. For this purpose, we intend to use the C library functions setjmp and longjmp (see also [Cameron et al., 1992]). In paragraph 7.5 another platform-independent implementation that uses the gcc extension of nested functions will be briefly described. A nonportable implementation, with less overhead and some additional features will be discussed in Chapter 8. Before we start implementing, however, we should first address some issues concerning the proposed syntax.

7.2 A simpler syntax

The first XTC implementation deals with almost all the syntax features, discussed in Chapter 6. The only exceptions are the way arguments are passed to the handlers and the way in which nesting is implemented. We will briefly discuss these two issues in this paragraph.

7.2.1 A different way of passing arguments in the first compiler

Example 6.1 shows that the way in which arguments are passed to exception handlers is (and should be) very similar to passing parameters to functions. Ideally, we want to be able to pass any type of argument that is also allowed in function calls. Moreover, there should not be much difference whatsoever between raising exceptions and calling functions.

For the first prototype of the XTC compiler, however, we will restrict ourselves to the passing of integers. Not only that, the format in which these integers are passed to the handlers will also differ somewhat from the syntax proposed in paragraph 6.2. It will look very much like the passing of parameters to main, i.e. using two parameters:

1. a pointer to the first element of an array of argument values (like argv),

29
2. an integer indicating the number of arguments (like `argc`).

This means that in this implementation the `raise` statement and the definition of the exception handler are as follows:

```c
raise e1 (a, b, c); /* a, b, c should be integers */
...
except e1 (int argc, int *args)
{ ... }
```

Even though this implementation is quite flexible and very usable as well, we do feel it ought to be modified to a more "function call"-like syntax. The reason for not using the proposed syntax in the first place is that the above syntax is much easier to type check at compile time and because at the time of designing the first XTC compiler there was no full syntax yet. The reason for not having changed it so far is that we wanted to get the second implementation of the compiler (the one we think more promising) to work flawlessly first. As will be shown in Chapter 8, this version will be able to cope with the proposed syntax. And not only for integers either, but for complex types and pointers as well. Overall, this way of argument passing should be regarded as a preliminary test to show that it could be done in a structured way.

### 7.2.2 Nesting

We think nesting of try blocks in one way or another should be allowed. For a start we will be content with just allowing functions with their own try blocks to be called inside a try. Eventually we would like to implement explicit nesting as well. In fact, what we really want is the try to be an ordinary compound statement. This means that try statements should be permitted to occur anywhere where a normal statement is also allowed (see paragraph 8.2.2). For the time being, however, a nesting based on function calls will suffice.

Note that the expressive power in such a scheme is the same as when it is possible to explicitly nest try blocks. It also implies that inside a function the try blocks are flat. Furthermore, since functions cannot be nested in C and are all defined at top level, the whole try structure is, in itself, flat. The advantage of such a property is that the task of building a `try tree` (a tree containing the hierarchical structure of try blocks and exception handlers for one function) is relatively easy to construct. The compiler only needs to make an entry per function for each try block. This is easy, because each function has at most a sequence of try blocks to store, (each of which may in turn have multiple handlers). A non-flat structure will be implemented in Chapter 8.

### 7.3 Handling the exception

Given the syntax of paragraph 6.2, we will briefly demonstrate how, in general, an exception is handled. This will help us to understand the implementation described in this thesis a little better. Consider the example shown in Figure 7.1.

In this example `main` starts a `try block` and in this block the function `recur` is called. Both in the prototype and in the function definition it is declared that `recur` may spread the contagious exception `e1`. The function `recur` then enters into a recursively nested call chain
void recur (int i) raises (e1); /* prototype */
int main ()
{
    try {
        recur (3); /* call function */
        except e1 (int a)
        {
            printf ("e1 was raised with arg %d\n", a);
        }
    }
    void recur (int i) raises (e1)
    {
        if (i == 1) raise e1 (i);
        else recur (i-1);
    }
}

Figure 7.1: Recursive function is called inside try block

of depth 3, at which point it raises the exception e1. There is no handler for such an exception in recur itself (in fact there is not even a try block) and the exception is therefore passed to the next level up (which is main), where it will be handled.

This is the external behaviour as observed by the user. In reality, however, the exception handling mechanism walks down the call chain step by step to see if there is an appropriate handler available. The first handler that matches e1 is the one in main, so this is the one that is executed. The whole process of handling the exception is illustrated in Figure 7.2.

7.4 Implementation: saving and restoring contexts

7.4.1 The first XTC compiler

We have seen what the compiler is supposed to do and what sort of code it is to handle, so it is now time to have a look at the first implementation. This implies that there is another implementation, that approaches the problem from a slightly different direction, and which is introduced in Chapter 8. In fact, we will sketch a third approach in paragraph 7.5. This implementation, which uses nested functions, however, was never implemented as a compiler.

The first XTC compiler is an attempt to arrive at a portable implementation by only using standard C statements. It is inspired by the preprocessor, described in [Roberts, 1989]. It does all the checking on syntax and correct use, that is required and translates exceptional C to machine independent standard C. However, because we started working on the next version, it does not completely implement the ideal syntax as mentioned in paragraph 6.2. Explicit nesting, for example, is not allowed and the passing of arguments is also not as good as we want it to be (only passing of integer values is allowed). The compiler is complete, however, in the sense that it checks whether users obey the (slightly restricted) syntax. Wherever in this implementation the syntax does not follow the proposed convention, we will explicitly say so.
7.4.2 Extending an existing C compiler

The main purpose of the XTC compiler is to translate exceptional C code to standard C. An additional feature would be to compile the code all the way to an executable. All things considered, the XTC compiler is just a standard C compiler which should be able to compile any C program whether it uses exceptions or not. If the program does use exceptions, we make XTC also produce a new source code in which the exceptions are translated to standard C statements.

For implementing the compiler we have used the well known GNU C Compiler (gcc, version 2.5.7), which was extended with a mechanism to check the syntax and correct use of exceptional C and a mechanism to replace the try, hit, finally and except statements in the original source file with the corresponding C code.

Another option would have been to start from scratch and just build our own preprocessor (which, for lack of time, is what we have done in the later implementation without setjmp). In that case it would be only a preprocessor, there would be no complete C compiler. The building of a complete C compiler is an enormous task, which is, even in the best possible conditions, way beyond the scope of this project.
Even building a preprocessor from the ground up proved to be very inconvenient. The problem is that, if you want to implement the compiler as an integral part of C (which exceeds the simple searching and replacing of a couple of macros), you need to recognize the entire C syntax anyway. Moreover, you have to build and maintain a substantial syntax tree which is very similar to gcc's, making it convenient to use that one.

The downside of extending the existing compiler, which involves a great deal of hacking in someone else's code, is that it is probably a bit more difficult. You are working with code of which you have only limited knowledge, which means you will have to spend a lot of time figuring out how certain things work, whereas if you write your own code you fully understand the meaning and behaviour of each function and statement. Eventhough the resulting compiler may be less powerful you are likely to get it up and running more quickly.

Despite the observed difficulties we decided, for the first implementation, to extend the existing GNU C Compiler (gcc). We don't think a mere preprocessor, built from scratch, is able to do all the checking on syntax and correct use of exceptions, that can be done with a complete compiler. The consequence of implementing the extension in the normal compiler itself is that the file that is being compiled has already been preprocessed. This means that all include files have been included, all macro definitions have been expanded and so on. It also means that all comments in the original code have been deleted, in short, it means that the generated C code will be less readable than the original.

Having said that, there is no need to really worry about it. The output file that is generated is not supposed to be readable. Why would anyone want to read that code, when there still is the original source? Programmers are not going to be very concerned about it, as it is compiler input only. An attempt will be made, however, to keep the output program as readable as possible, just in case, but as long as the original file is clear enough, we don't think a result which is a little less readable is much of a problem.

### 7.4.3 Recognising and checking exceptional C

One of the most important features of the XTC compiler is its ability to recognize the exceptional statements and to generate error messages if there is something wrong in syntax or use. This is not as simple as it sounds, because it means checking on the existence of handlers for all functions that may possibly raise exceptions, checking whether excepts and finally's are not used at the same level (once a finally block is finished it can only raise or reraise exceptions at a higher level than its own), checking whether contagious exceptions in function prototypes and definitions match, checking number and type of arguments, and so on.

Summarizing, the compiler will generate usage errors in the following cases:

1. **errors in contagious exceptions list**
   - exception name in prototype has no match in definition
   - exception name in definition has no match in prototype
   - no contagious exceptions allowed for "main" (skipped)

2. **absence of handlers**
   - exception of function name cannot be deferred
   - no handler for exception name of name
   - invalid function call name

---

Exceptional C: Design and Implementation of the XTC Compiler
3. errors in argument passing
   - first xption argument must be an (argument count) integer
   - second xption argument should be pointer to int
   - exactly two args in exception definition right now
   - only integer arguments allowed
   - too many arguments for exception (max: \(\text{MAX} \cdot \text{ARGS}\))
   - too many arguments defined in exception handler

4. other
   - nested try within same `try` not allowed (yet)
   - RAISE name outside try not contagious for function
   - this implementation of xtc needs args, sorry
   - FINALLY not allowed outside TRY
   - illegal combination of EXCEPT and FINALLY
   - FINALLY already exists in this try
   - EXCEPT statement not allowed outside TRY

Strictness
At this moment the XTC compiler keeps track of all these things and produces corresponding messages in case of errors. The compiler never aborts, however, at least not for errors in exception syntax. One last remark before we go and look at the datastructures necessary for doing all this error detection: the XTC compiler is very, very strict in the sense that it generates errors for every occurrence of a function that may `raise` a certain exception and defer it to a higher level and for which there is no handler at this higher level.

Many implementations do not share this point of view. They are quite willing to allow exceptions to be raised for which there exists no appropriate handler. Such a raise generally results in termination of the program in a state suitable for debugging. We think it does not have to come to that. We don’t want the program to crash and we don’t `have to` let it crash. If the compiler detects in a program the absence of a handler it marks this code as illegal and the programmer will just have to do better. This means that there will always be an appropriate handler for every exception raised.

7.4.4 Checking the input code: data structures
We will now discuss the inside of the compiler. How does it check on correctness of use and what sort of data structures are needed to do so? When the XTC compiler parses the input code (i.e. an exceptional C program), the whole program is parsed in one go, reading all the functions, statements, expressions and other program text sequentially. In this context we define “current function” as the function which is being parsed at this moment (“current try block” will be defined likewise, and so on) [Bennet, 1990].

Linked lists of contagious exceptions
Whenever a function prototype or a function definition is encountered, a new `function` entry is added to a complex tree structure, which keeps track of all exception-related program code in each function. At the highest level of abstraction this structure is really a linked list of all the functions in the program.
Part of this tree (at a lower level), and as such linked to each entry that represents a function, will be another linked list, consisting of all contagious exceptions for this particular function. The structure of such a list is illustrated in Figure 7.3.

![Figure 7.3: Linked list of all contagious exceptions for one function](image)

Whenever the compiler comes across a function prototype or definition, it grabs the “raises” list of contagious exceptions (if any), builds the corresponding list and links it to the function. The contagious list in prototype and definition will be expected to match and the compiler checks to see if they do.

Now, when a certain function is called somewhere in the original program text, this function is either defined already or a prototype of the function was parsed already (property of C). In either case it will be known *apriori* which exceptions are contagious for this function. Therefore, a raise statement outside a try block can be easily checked: the exception should be contagious for the current function as well. Raise statements inside a try block prove a bit harder to check (because it may have a handler associated with this very try block as well) and will be dealt with next. First, however, we need to discuss the datastructures needed for this task.

**Try tree datastructures**

Obviously, the compiler needs to keep track of all the try blocks and their corresponding handlers. Therefore, for each function in the tree structure, which we will henceforth call *global exception tree*, we will maintain a data structure, that represents all of its try blocks and their corresponding exception handlers. It can easily be shown that this structure is a subtree in itself. Such a subtree will be called a *try tree*. It consists of a linked list of try blocks, each of which is connected to yet another linked list (the *handler list*), consisting of all the exception handlers associated with that try block. In fact, the first handler of the list is entered in the node for the try block itself. A graphical representation of a try tree is given in Figure 7.4.

Figure 7.4 is the representation of a function containing three sequential try blocks. The first try block is endowed with four exception handlers (with names “xption 1.1”, “xption 1.2”, “xption 1.3” and “xption 1.4”, respectively), the second try contains two handlers and so on. The figure shows that there is no possibility to explicitly nest try blocks, otherwise it would not have been a linked list of try blocks, but a proper tree. We will see an example of such a tree in Chapter 8, where we *will* allow try blocks to be nested in other try blocks (and even in exception handlers).
Figure 7.4: Try tree for one function, containing all try blocks and their associated handlers

Summary: global exception tree datastructure

Summarizing, the global exception tree consists of the following (see also Figure 7.5):

- a linked list where each node represents a function in the program,
- links for each function, attached to:
  1. a linked list of all the contagious exceptions that this function may spread,
  2. a try tree structure, consisting of:
     (a) a linked list of all try blocks in the function
     (b) with attached to each try block a linked list of handlers associated with it.

Correct use of raise in a try block

We now return to the issue of checking the correct use of raise statements that occur inside a try block. The XTC compiler checks the use to the extent that it will generate errors if no handler has been provided for a certain exception that may be raised. We can only say the raise statement (or the function that may spread the contagious exception under consideration) is illegal when the following conditions are both true:

1. the exception is not contagious for the current function
Figure 7.5: Global exception tree datastructure: all exception related information per function

2. the exception has no handler associated with this very try block

However, the latter can only be checked after the whole try clause has been parsed and is not known beforehand. We therefore enter undeferrable raises inside a try clause in a special data structure called pending_defers (because it contains all the exceptions that were deferred either by functions at a lower level or by explicit raise statements at the current). Now the rest of the try block is parsed. Whenever the compiler finds an exception handler, an entry is made in the handler list of the global data structure (i.e. it is entered in the try tree of the function to which it belongs).

When the end of the try block has been detected, all exceptions in the pending_defers data structure are checked once again to see whether or not there exists a handler for them in the current try.

Syntax checking is static by nature: we only look at the program text when it is being compiled and we immediately generate error messages. Closely related to this static checking is what we call static translation, which consists of such things as replacing the contagious

---

Exceptional C: design and implementation of the XTC compiler
exception list in function prototypes and definitions with normal C comments, saying which exceptions may be raised by this particular function. For example:

```c
void foobie (int i) raises (e1, e2)
{ }
```

will be translated with:

```c
void foobie (int i)
  /* exceptions: e1, e2 */
{ }
```

The static translation has no consequences at execution time. It mainly consists of entering C comments in the program text to clarify certain things.

### 7.4.5 Handling exceptions: the code

So far we have seen the external behaviour of the XTC compiler with respect to the way in which it recognises and checks its syntax. The actual implementation of the exception handling mechanism will be discussed next. With implementation in this context we mean the translation of exception statements and handlers to C. We will find the implementation to be rather complex, so we will introduce key components and functions step by step.

We should keep in mind that this part of the compiler deals with what we will call dynamic translation. As opposed to the static syntax checking (and static translation) where we only extract information out of the program text once (e.g. about existence of handlers, illegal contagious exceptions, etc), we now have to introduce program code that will be called at run time, i.e. each time a specific function, try clause, hit statement, etc., is executed. When a try block is placed in a loop in the original program and the statements in the loop are executed say a hundred times, this means that for each of these times the dynamic translation of the try block will be executed as well.

The opposite of dynamic translation, by the way, is of course the static translation of the previous paragraph, which involved such things as replacing the contagious exception list in function prototypes and definitions with normal C comments, saying which exceptions will be deferred by this particular function.

Note that even in dynamic translation the output code is generated at compile time. The only difference between static and dynamic in this context is that the dynamic translation has severe consequences at run time. Here are all the statements that will eventually be executed.

**Rough sketch of implementation**

Of considerable importance for our implementation is the notion of a context block. A context block is a structure that corresponds to one single (active) try clause. As we will show next, this is where we will save our current context in (registers, etc) and also some additional information which has to do with exceptions. Saving contexts is done by using the ANSI C library routine `setjmp` [Kernighan and Ritchie, 1988]. A linked list of context blocks can be regarded as an exception stack (see Figure 7.6). The pointer that points to the head of this list is therefore called the exception stack pointer. A context block has a field for:

1. the context of a try clause (of type `jmp_buf`)

---

**Exceptional C: design and implementation of the XTC compiler**
2. the name of the exception that is raised
3. the arguments that will be passed to the exception
4. the number of arguments passed to the exception
5. an indicator flagging amongst other things whether or not there is an exception raised in this try clause, which is in the process of being handled somewhere (possibly outside this try)
6. a link to the context block of the try clause at the next level up

Figure 7.6: Linked list of context blocks for each try

The general idea of the way in which exception handling is implemented is as follows (see also Figure 7.7). We assume the program is in the middle of execution and is about to enter a try clause.

1. When try is detected in the program code we save our current context in a context block, using the setjmp command. The context block was newly created for this try. Most fields of the context blocks, except the first, are relatively unimportant at this moment, because no exception has been raised yet.

2. Next the try statements are executed (i.e. the statements between the try keyword and the first exception handler).

3. If no exception was raised at all, the try clause ends and the context block is destroyed, nothing special here.

4. If there was an exception raised somewhere (the interesting case), then the following actions are initiated:

- Starting with the last one (i.e. the one at the head of the list, pointed to by the exception stack pointer) we walk down the linked list of context blocks; in this current block there is a flag indicating amongst other things whether or not an exception is already raised in this try clause (such a condition will turn out to
be only possible if there are nested try blocks). We walk through this chain of contexts until we find a block where there is no exception raised yet.

- Since there is no exception active (raised) in this context, we can now:
  
  (a) enter the name of the exception that was raised just now in the appropriate field of the try clause's context block,
  
  (b) enter the arguments and number of arguments in their appropriate fields as well,
  
  (c) restore the context of this particular try clause by making a longjmp to the context in its context field.

5. Back in this context the handlers in the try clause are investigated to see if one of them matches the name of the exception in the context block.

6. If so, the handler code is executed.

7. If no handler matches the raised exception, the exception is re-raised with the same name and the same arguments and the process is repeated until an appropriate handler is found.

8. If at the toplevel we still have not found a corresponding handler for the exception, the programs halts using abort() (with the strictness of the compiler as it is now, this should never happen, because an absent handler causes } to generate an error).

### 7.4.6 Core of the handling mechanism: setjmp/longjmp

It should be clear that the exception handling mechanism is built around the library functions setjmp and longjmp [Kernighan and Ritchie, 1988]. Setjmp just grabs all the relevant registers (and on Unix the state of signals as well) and saves them in a buffer of the predefined type jmp_buf. After the setjmp instruction has saved the context, it returns with the value 0.

Then, the program continues its execution and nothing special happens until a raise instruction is executed. Now, we make a longjmp to the first available context block. Longjmp restores the context that was saved in that context block and literally jumps to the place in the code where the setjmp instruction that saved this particular context was executed. Program execution will now continue in such a way that it looks as if we had executed this setjmp instruction just now, with the exception that it has now returned a value other than 0!

So far, we are still ignorant of the fact whether this block has an appropriate handler for us or not. This is only checked after the jump is made. If it turns out that this context cannot help us either, another longjmp is made, and so on. In C, the translation of a try block with associated handlers would therefore roughly look like:

```c
if (!setjmp (new context block jmp_buf))
{
    /* try statements */
}
else if ( name of handler matches raised exception )
{
    /* handler statements */
```

---

**Exceptional C: design and implementation of the XTC compiler**
Suppose we are executing this program and program execution has just called `func2()`.
We have then entered 2 try blocks so far (the one in `main` and the one in `func1`). This means the list of context blocks should look like the one depicted on the right:

At this moment the following piece of code is executed:

```c
void func1(): e1
{}. try
{}  { func2();
{}      raise e1 (1);
{}  }
{ }
```

because no exception has been raised yet, "e1" and its args are entered in this context block.

Figure 7.7: Handling an exception: implementation
Similarly, a "raise name (args) statement" for this handler would be translated in the following fashion:

```c
store_in_context_block ( exception name and arguments );
longjmp ( jmp_buf in the context block );
```

The advantage of such a scheme is the portability of the generated code, because `setjmp` and `longjmp` are machine independent library functions. This is good, it means that we have a general solution to the problem of exception handling. Moreover, it was mentioned earlier that we even have a complete compiler for the exceptional C code (albeit with a slightly different syntax). Not only do we have a general solution, but we are also able to run the programs that are generated on any machine we like.

The major disadvantage however, is the poor performance. The `setjmp` instruction is very slow. Considering it has to be executed for every try block in our program, such a mechanism totally obliterates the goal of "a low overhead scheme in case no exceptions are raised". Instead, we have a very high overhead implementation, regardless of the fact whether exceptions are raised or not. Since we assumed exceptions to be rare, this problem is significant enough to justify our having second thoughts about the `setjmp` function. In fact, this significance can be expressed in a performance which is reduced by at least a factor 100, if we decide to use try blocks (see paragraph 9.2). Such a performance loss is sure to discourage programmers to use the facilities.

The conclusion is, therefore, that the idea is viable, but does not achieve all the goals that were set. It is certainly worth our while to look for a better solution, resulting hopefully in an implementation that incurs only little overhead in the normal case (when no exceptions are raised). This problem will be addressed in the next chapter where we will abandon the `setjmp` altogether. In the next paragraph we will describe another portable solution for exception handling, that does not use the `setjmp` instruction either.

### 7.5 Portable exceptions using nested functions

#### 7.5.1 Introduction

The `setjmp/longjmp` implementation is not the only approach to arrive at a portable exception compiler. We will now describe another method that uses the `gcc` extension of nested function definitions to deal with exceptions. For lack of time, we have not implemented a compiler for this approach, but the idea proved to be too interesting to be ignored. We will discuss some of the issues at stake in this implementation, as well as some advantages and disadvantages. Although we do not have a compiler for it, we have built a few test programs manually. Building a compiler for it, is relatively straightforward. It should recognise the syntax, discussed in Chapter 6, check its correctness and, finally, translate it to standard C, using the data structures of paragraph 7.4.4. The performance of these programs will be compared with the performance of the other implementations in Chapter 9.
7.5.2 Nested functions and exception handling

The GNU C compiler gcc provides several language features not found in ANSI standard C, some of which are very powerful. Nested function definitions, for example, are foreign to ANSI C, but in gcc you can explicitly define a function inside any other function (including nested functions). This is what we call *lexical nesting*. The nested function’s name is local to the block where it is defined. The nested function can access all the variables of the containing function that are visible at the point of its definition. This is called *lexical scoping*. Nested function definitions are permitted within functions in those places where variable definitions are also allowed; that is, in any block, before the first statement in the block.

Furthermore, it is possible to call the nested function from outside the scope of its name by storing its address or passing the address to another function. Gcc implements taking the address of a nested function using a technique called *trampolines* (see also [Brue, 1988]). The last property of nested functions that is very useful for our exception handling mechanism is that a nested function can jump to a label inherited from a containing function, provided that the label was explicitly declared in the containing function. Such a jump returns instantly to the containing function, exiting the nested function which did the goto and any intermediate functions as well. This is very powerful, because in this way the jump instruction automatically unwinds the stack.

For example, consider the following program in exceptional C:

```c
bar () raises (e1) /* the raising function, called by "foo ()" */
{ raise e1 (args); }

foo ()
{
    try {
        bar ();
        except e1 (ArgType arguments)
            { do_something (); }
    }
    code_outside_try ();
}
```

As a first step in the implementation of a portable exception handling mechanism using nested functions, we now consider the following program:

```c
void (*trampoline) ()
{

foo () /* foo contains a handler "e1" */
{
    __label__ EndTry; /* gcc's way of declaring a label explicitly */

    e1 (ArgType args) /* handler implemented as nested function */
    {
        do_something ();
        goto EndTry; /* jump outside function */
    }
    trampoline = e1; /* set up trampoline function (on stack) */
```

---

**Exceptional C: Design and Implementation of the XTC Compiler**
bar ();

endtry:
    code_outside_try ();
}

In this program a trampoline function is set up, and instead of calling “e1” directly, a call is
made to to the trampoline. To raise exception “e1”, all bar() has to do is to call trampoline
(args). The nested function will be executed and instead of resuming bar as in a normal
function, we jump to the end of the try block. In fact, at this point we could even decide to
resume the raising operation, by not executing the jump. In that case the function would
return normally and execution of bar () would be continued at the next statement. We have
already pointed out that, for a number of reasons, resumption is not a very desirable property
(see also paragraph 2.3.2). But even if it would have been advantageous, we will show that
the solution above is incomplete and would incur a lot of overhead. To reduce the overhead
we will (have to) change the implementation in a way that leaves no room for the resumption
model.

The passing of arguments in a “function like” manner, also seems very straightforward in
this solution. Raising an exception is not just very similar to a function call, it is a function
call. However, for this property also holds that it will be removed, when we change the
implementation to reduce the overhead. In the next paragraph we will illustrate the overhead
problem.

7.5.3 Issues and problems

The example translation using nested functions on page 43, is clearly not complete. It uses
only one trampoline function, that is declared at the top level. If a programmer decides to
nest try blocks, this solution will not suffice. Instead, we will need a proper exception stack,
that keeps track of the appropriate nested function (the handler) for each exception that may
be raised. If there would be only one exception in the entire program, such a stack could be
a linked list, where each node pointed to the appropriate trampoline function to execute if
the exception is raised (the last node in the list corresponds to the current position in the
program, the previous node corresponds to the next level up, and so on). Because we allow
more than one exception to be defined in a program we will have to expand the linked list
to a tree structure. Note that such a stack would have to be maintained at run time. Each
time a handler is encountered at runtime, we will have to update the exception stack (i.e. set
up the trampoline for the nested function, and push a pointer to it on the exception stack).
This is not a trivial task, especially when we are dealing with default or finally handlers that
must be inserted in all the subtrees.

Overhead

Although the mechanism described above seems to work fine, it is not as good as one might
think. Setting up a trampoline takes quite some time and on top of that the exception stack
has to be maintained. Both operations have to be performed at run time, for each exception
handler that is encountered. If the try blocks contain more than a couple of handlers, this
may well make the implementation even slower than when a setjmp instruction is executed.
Also, because the handlers are now function calls, we will have the additional overhead of the

Exceptional C: design and implementation of the XTC compiler
call, for all finally handlers as well, even when no exception is raised (small finally handlers can, of course, be inlined). To make the solution useful, we will have to modify the code in such a way that the overhead will be acceptable.

### 7.5.4 Nested functions exception handling: implementation

Because we have no use for a mechanism with potentially so much overhead, we will modify the original translation in the following way. First of all we will only keep one nested function for each try clause (and thus one trampoline on the stack). This function will handle all exceptions that are raised in the scope of the try. A switch statement is used to determine the appropriate handler. The nested function is kept as small as possible to make sure it can be inlined. For this purpose we move the switch and the actual handlers outside the nested function, into the containing level.

The switch has a default, that can be used for default or finally handlers, but if no such handlers exist, we will still have to provide a standard default for it. The standard default is only executed if none of the cases matched the exception, so we need to re-raise the exception here, to pass it on to the next level up.

An additional advantage of the “one function for all exceptions” approach is that the exception stack is reduced to a linked list again, making it easy to be maintained. All a node in the list needs to contain is a pointer to the nested handler function and a link to the previous node.

Without too much explanation, we will give our new solution for exception handling based on nested functions. The only assumption is that raised exceptions are identified by a number “Xption_id” and at compile time we will assign a unique identifier to each exception (for example, the exception “e1” will be identified by the the integer number e1, that represents the string). The translation of the exceptional C program on page 43, would look like the following:

```c
[1] void foo ()
[2] {
[3]     /* begin try block */
[4]     __label__ _HANDLERS_
[5]     _label__ EndTry;
[6]     int Xption_id = 0; /* used in switch (exceptions are identified by a number) */
[7]     int *Argp; /* pointer to args */
[8]     Exception_S{mp p; /* exception stack struct */
[9]     void _handle_all_ (int exception_id, int* args)
[10] {
[12]     Argp = args;
[13]     goto _HANDLERS_
[14] }
[15] }
[16] p.handler = _handle_all_; /* here we push the handler on */
[17] p.prev = ExnS{ackp; /* the exception stack */
[18] ExnStackp = &p;
[19] bar (); /* all user code comes here */
[20] ExnStackp = ExnStackp->prev; /* pop handler off exception stack */
[21] goto EndTry;
[22] _HANDLERS_
[23] switch (Xption_id) {
```

---

**Exceptional C: design and implementation of the XTC compiler**
case e1: { unpack_args (Argp); do_something (); break; }

default: ... "reraise exception" ...

EndTry; 

); }

} code_outside_try();

}

It should be clear that if the program contains another exception, for example e2, the switch statement would have contained a "case e2:" as well. Lines 4-8 initialise the variables that are needed for a try clause. The reason for declaring variables “Xoption.id” (identifying the exception raised) and “Argp” (a pointer to the location where all the arguments are stored), which are global to the handler function, is that we need to access their values in the switch statement and the handler respectively. The handler definition starts in line 10. The goto in line 14 is a sort of stack unwinder. It jumps to the switch statement in the containing function and the corresponding handler will be executed. The goto in line 24 makes sure that the handler code is skipped when no exception is raised.

The passing of arguments will not be dealt with in detail. We can implement it using the argv, arg variables of the setjmp compiler, or we could try and program it in a more “function like” manner (clearly, this is no trivial task). Note also, that in the solution shown, there is no longer the possibility to resume the operation that raised the exception.

7.5.5 Summary

A new approach towards exception handling is the use of nested functions. Although no compiler has been built for it, we have demonstrated the implementation of such a mechanism in a way that keeps the overhead as low as possible (although it cannot be made entirely overheadless). The advantages of this scheme are:

- Easy to implement. The idea itself is very straightforward and we have shown that the implementation is simple and elegant.

- No need for setjmp.

- Portable code. This is clearly the greatest advantage. Although faster implementations are possible (see Chapter 8), they generally depend on properties of the architecture to work correctly. This solution will run on any machine and on any processor (provided that gcc is installed on it).

- Lexical nesting. Obviously the technique is quite suitable for explicit nesting as well. Nesting of try blocks is easy, because we can just define new nested functions.

- Most optimisation is possible. Optimisation will be discussed in more detail in Chapter 9.

- Fast raising. The raising of exception is relatively fast, because it only involves the packing of arguments and a function call.

The disadvantages all amount to the problem of the overhead that is introduced by the mechanism again. Things seem to be a bit better than in the setjmp implementation, but we should be careful with statements like that, because overhead is incurred by quite a number of causes:
7.5. PORTABLE EXCEPTIONS USING NESTED FUNCTIONS

- **Initialisations.** At the start of the try block we declare and initialise some variables. Although this does not take much time, the consequences will be significant, when we put *try blocks* in tight loops that iterate many times.

- **Setting up the trampoline.** For each try clause a trampoline function has to be set up. A pointer to it is pushed on the exception stack, but that is of no importance at this point. Setting up a trampoline may be relatively slow, especially on machines with separate caches for instructions (I-caches) and data (D-caches), like the DEC Alpha. On these machines the I-cache will have to be flushed each time the trampoline is set up and the stack has to be made executable. For this purpose a system call is needed (at least it is on the Alpha), which slows things down considerably.

- **Maintaining the stack.** We can split the overhead caused by the exception stack in two operations:

  1. *pushing* a handler, which involves making a new block on the exception stack, storing the address of the trampoline in it; linking the new block to the existing stack (one assignment) and making the exception stack pointer point to the new block (another assignment).
  2. *popping* a handler, which only involves one assignment, to restore the exception stack pointer to the previous level.

- **Overhead for finally handlers.** The execution of a *finally* is not necessarily caused by a raised exception. If a *finally* is executed in the normal case, we want it to incur no overhead (or very little). In this implementation, however, we always have to do the function call to the trampoline, regardless of the fact whether exceptions are raised or not. Even if the function itself is inlined, we still have the two assignments in the function that will be executed (or some other code, depending on the implementation).

For lack of time, no compiler was built for this implementation. We do believe, however, that this approach is promising and some further research should be devoted to it. It offers a *portable* solution to the exception handling problem in which we no longer have to use the inefficient *setjmp* instruction. Having said that, we should not forget that there is still quite some overhead involved with the implementation. We feel it is a good solution, but certainly not the *ideal* solution. Especially if *try blocks* in a tight loop would be lexically nested, all the pushing and popping would take a lot of time, as will be shown in Chapter 9. The ideal solution would do without this overhead altogether. It means we still have to look for a faster solution, and for this purpose we will have a look at a nonportable implementation.
Chapter 8

XTC-2: low overhead exceptions

8.1 Introduction

The two designs in the previous chapter showed that an exception handling mechanism in C is feasible. We have implemented a compiler that covered most of the issues relevant to exception handling. It failed, however, in yielding code with a low overhead. The use of try blocks, to make programs reliable, is severely punished by the collapsing performance. The prospect of putting these try blocks in a loop, which iterates a great number of times, is quite dreadful. We also demonstrated that the cause of the overhead is the context saving, that is done each time a try is entered. In the last paragraph, we described another portable solution to the exception handling problem. Although the *setjmp* instruction has disappeared in this code, we also showed, that there is at least some overhead, and that it may be considerable. For lack of time, no compiler is built for that solution, but we feel it is interesting enough for further experiments.

In this chapter we shall try a different approach. It incorporates the replacement of the inefficient *setjmp* instruction by a genuine stack unwinding routine. Quite a number of features of the first design, however, survived in the new version, sometimes in a somewhat enhanced form. We will also find that XTC-2 is able to cope with a more complex syntax. We will address the topics related to the syntax first.

8.2 Syntax issues

One of the properties of the *setjmp* design that clearly needs improvement is its inability to deal with all the aspects of the proposed syntax for exceptional C (see paragraph 6.2). Although most of the syntax agrees with the compiler just fine, it is rather limited in the way in which it allows nesting of try blocks, as well as in the way in which arguments are passed. In the new design, we want to abandon those restrictions, so that we can make full use of the features that exceptional C provides. The main issues at stake are concerned with nesting and argument passing. They will be addressed separately next.

We must keep in mind that for the time being, we will not build a complete compiler, but merely a preprocessor. This implies that we can neither completely check the correct use of exceptions, nor whether the types of arguments passed to the handlers correspond with the format in the handlers’ headers. We do feel that an implementation in gcc itself is not only useful, but even essential if we are to use this exception handling mechanism on a large scale.
For this reason, the preprocessor stays as close to the full grown compiler as possible. For instance, it builds all the necessary data structures (like the one in Figure 7.5, but with a more complicated structure), to check correctness. Strictly speaking, we can make the preprocessor work without the data structures as well. Maintaining a well-structured exception tree, however, will enable us to integrate the whole implementation with gcc, without changing much in the code. Also, a data structure like the one we are using is relatively easy to extend, if need be. The new data structures will be dealt with in paragraph 8.2.2.

8.2.1 Passing arguments

This time we should make an attempt to make the passing of arguments to exception handlers more "function like". The previous way of doing it, using an argc and argv parameter (very much like the ones used to pass arguments to main), provided an elegant way of passing an arbitrary number of arguments to an exception handler (which we thought to be important at the time), but it only worked if the arguments were of type integer. Moreover, it failed to provide a uniform way of passing arguments.

Although the mechanism could easily be extended to incorporate other types as well, this is really not the way to go. What we want is exception definitions which look exactly like function definitions, with the sole difference that it is preceded by the keyword except. This means that many different types can be passed which are declared in the exception header, as illustrated in Figure 6.1. Likewise, raises should be allowed to pass more or less any type that is available without having to do any type casts.

8.2.2 Explicit nesting

The second syntax feature that we want to incorporate is explicit nesting. The previous design only allowed for nesting by way of function calls (we ignore the solution using nested functions, at this point, because it was never implemented as a compiler). It would be far better, if we could just start new try blocks inside other try blocks and so on. In fact, we would like to look at try blocks as normal statements. This means that we are allowed to define try blocks inside exception handlers or even inside finally handlers (see Figure 8.1).

The latter case is particularly interesting. Suppose for example, that the finally handler is executed on account of an exception \( \varepsilon \) and that it enters the try block contained in its body. Inside this try block we raise a second exception \( \xi \), for which a handler is provided in this very try clause. The \( \xi \) handler will subsequently be executed and the inner try ends. In this case the exception handling mechanism has to make sure that, at the end of the finally, exception \( \varepsilon \) is reraised. Thus, the explicitly nested try statements make the implementation a bit more complicated.

Data structures

Before doing the actual translation, the XTC preprocessor first reads the exceptional C program code and builds data structures, that are similar to the ones built by the full grown compiler (see Figure 7.5). Doing this guarantees that the resulting code can be easily incorporated in gcc itself. An additional advantage is that it is easier to do processing on such a well-structured tree than it would be on flat code.

The data structures in our new design turn out to be a bit more complex than the ones of the previous chapter, as was to be expected, because of the explicit nesting. Explicit nesting
int main ()
{
    try {
        printf("try clause 1\n");
        try {
            printf("try clause 2\n");
            raise e1 (1);
            finally {
                printf ("finally handler\n");
                try {
                    raise e2 (100);
                    printf ("try clause 3\n");
                    except e2 (int  a) {
                        printf ("handler e2\n");
                    }
                }
            }
        }
    }
    except e1 (int  a) {
        printf("got here by a reraise of ‘e1’ in the finally\n");
    }
}

Figure 8.1: A complex example of nested try statements

(or rather: regarding a try block as an ordinary statement) compels us to add a new field to each entry in the try tree of Figure 7.4. Obviously, since try clauses may contain other try clauses, we need to extend the block for try clauses (the so called try nodes) in this structure with a link to the nested subtree (if any). Likewise, since handlers can also contain try clauses in their bodies now, these structures should also be extended with such a link. An example of the resulting try tree structure is depicted in Figure 8.2. In this tree the try nodes have two fields with which they can be linked to other try nodes (the fields of try nodes as well as those of handler nodes will be discussed in paragraph 8.3.2):

- a “nested try link” field, pointing to the nested try clauses it contains,
- a “next try link” field, pointing to the next sequential try statement on the same level of nesting.

The program that corresponds to Figure 8.2 is rather complex. It contains both nested and sequential try clauses. Appendix A shows an example of the type of program that would produce a try tree like that.

8.3 Outline of the implementation

8.3.1 Stack unwinding versus context saving

The bottleneck in our previous design was the setjmp instruction upon entry of the try block, so the obvious solution is to get rid of that part of the implementation, i.e. the part that
needed to save the context, altogether. If need be, we could try and move some of the overhead to the raise statement. We have already pointed out that it does not matter if raises are slow, since they are not likely to happen very often. Note, that if no context is saved for a try clause using setjmp, we are also no longer able to restore the context in one go by simply executing a longjmp.

In our new compiler we will save no context whatsoever. Instead, when an exception is raised, we try to reconstruct the appropriate context. The appropriate context is the context (the right state of the registers etc.) of the try clause, or rather the exception handler, that will eventually handle the raised exception. This means we will need to dig our way through the stack, undoing all the actions that modified the registers until we find the right handler for this exception. Figure 8.4 illustrates this process.

8.3.2 Range tables

The basic idea for the implementation is relatively straightforward. What we need to do is determine at compile time rather than at run time, which handler should be associated with what piece of code. For this purpose, we introduce the notion of ranges and range tables. Consider a program that declares exactly one exception (say e1). If we only look at the flat structure of this exceptional C program, we notice that it can be subdivided in a number of disjunct areas, each of which is associated with exactly one exception handler. Note, that this exceeds the mere division of the program in try blocks, because try blocks need not be disjunct. We will call the disjunct areas ranges. Figure 8.3 illustrates how a piece of code can be chopped up in disjunct regions and also shows the handlers that correspond to each of these regions.
8.3. **OUTLINE OF THE IMPLEMENTATION**

If the program contains multiple exceptions the situation does not change much. For each exception individually, disjunct ranges can be identified. Ranges of different exceptions are allowed to overlap, of course, and indeed, they generally do. A range table for one single exception, will now contain tuples of three values, uniquely identifying all the disjunct ranges, together with their associated handler. The tuples will be of the sort:

\[ \text{(range begin, range end, handler)} \]

We furthermore define a handler table to be the set of all the range tables (one for each exception) for a particular program. Handler tables are very useful. They permit us to determine, by looking only at the position (the current instruction) where we are in the source code, which handler should be executed when a particular exception is raised. In fact, this is precisely what the exception handling mechanism will do, albeit in a slightly more complicated manner. If an exception is raised, it will take the program counter, and compare it to the ranges in the range table that corresponds to that exception. It will look for that particular range \( r \) for which the following holds:

\[ \text{range begin } \leq \text{ program counter } < \text{ range end} \]

Thus, it will find the range that contains the current program counter and that will point it to the address of the right handler. The implementation of this task is discussed in paragraph 8.4.4 and is illustrated in Figure 8.7. To facilitate the creation of range tables, we further embellish the try tree structure with fields for the indices of where each try statement begins, where it ends and where each handler begins. Of importance for the range table generation are the indices (locations in the code) that are listed below. Figure 8.5 shows a graphical representation of the fields of both try nodes and handler nodes.

---

**Exceptional C: Design and Implementation of the XTC Compiler**
here we enter try clause

this is the stack at handler point

stack frame 1 frame 2 frame 3

do all registers actions raise

STACK

Figure 8.4: Unwinding the stack when a raise is executed

1. **begin of try clause**, because this is where the scope of the handlers associated with it starts; we will call this the *try_begin*;

2. **begin of first “except” keyword**, because this is where the scope of the handlers associated with this try ends; we will call this position the *scope_stop*;

3. **begin of each handler**: we will call this position the *handler_label* (at a later stage we will replace the “except” keywords with real labels to mark their positions, hence the name), they are important when the range tables are created; the ranges for this try will be of the form *(try_begin, scope_stop, handler_label)*.

4. **end of try**, this is really nothing more than a mark that this try is properly ended.

Now all that remains to be done is unwind the stack and undo the register actions (Figure 8.4). What we effectively do, is build a context corresponding to the one that would have existed for the handler. This context will be stored in a normal *jmp_buf*, so that we can subsequently execute a *longjmp* to restore the context.

### 8.4 Implementation of the mechanism

#### 8.4.1 Building the try tree

Building a multidimensional data structure, like the try tree, out of an intrinsically flat file, needs to be done with some caution. The way the try tree is organised suggests that we should solve this problem recursively. Observe that sequential try statements are completely independent, in the sense that whatever is contained in the second try cannot influence the first try in any way (and vice versa). The first try will have been completely parsed by the time we arrive at the second. We can use this property to start building a new tree from the ground up, whenever we find a sequential try in the code. Note also, that no such thing holds for nested try clauses.
The actual construction of the tree will be done by calling the function `build_try_tree()`. We will only briefly discuss the way in which we do this. The basic steps upon detection of a try clause are the following (we assume that we already have a half-finished tree and now find a new try clause in the program text):

1. If the try clause is a nested try clause (i.e. if the last try that was parsed has not seen a proper end of try), a new try node is created and hooked to the “nested try link” field of the try clause that was last parsed. We keep track of the level of nesting (the depth, which is now increased by 1) where the try resides.

2. If the try clause is nested inside an exception handler, roughly the same actions are taken. In this case, however, the try node must be linked to the appropriate field in the handler node.

3. If the try was a sequential try clause (i.e. the last try parsed was properly ended), we just call `build_try_tree()` recursively. If this function returns, we hook the tree, that was produced by it, to the “next try link” field of the try clause that was last parsed. We have now dealt with everything that was contained in this try clause (including all the try clauses it contained, no matter how deeply nested). The level of nesting does not change.

The actions taken when the end of a try clause is detected are then:

1. If the last exceptional instruction that we detected was not an indication of end of try (which means that this level of nesting is not finished yet), mark this try as ended, but do not change the level of nesting.

2. If the last exceptional instruction that we detected was an end of try indication, we have now reached the end of this level of nesting (check!). There are two possible cases:
   - if we are not yet at the top level, then move up one level of nesting and continue parsing there (after marking the try corresponding to the detected end of try indication as properly ended)
   - if we have reached the top, we return a pointer to the root of the tree that we have built

Obviously detection of exception handlers leads to creation of new handler nodes, that are linked to the appropriate fields in either the try node (if this is the first handler), or the preceding handler node. It is also rather obvious that:

- whenever a try is detected, the field try_begin in the newly created node will be initialised with the current position in the program (this will turn out to be a label)
- whenever an except is found, the handler_label will be loaded, similarly, and also, if this was the first except in the try clause, the scope_stop will be initialised
- whenever an end of try is found, the corresponding field in the node for this try clause is initialised
**Summary**

We have now seen what a *try tree* consists of and how it is built. We have also seen that it contains all the information concerning where scopes of exception handlers start, where they stop, and where we can find a specific handler. Still, we are not quite there yet. The code areas that *try nodes* contains, for example, are not disjunct for the same exception. They merely indicate the scope of the associated handlers. We now have to extract the ranges, such that the following holds:

\[
\forall (\text{exception } \xi): \text{range}_i(\xi) \cap \text{range}_j(\xi) = \emptyset \quad (\forall i \neq j)
\]

Furthermore, the ranges have to be organised in tables, one table for each exception. The union of all ranges in the table for exception $\xi$ has to cover all the areas of program code that are in the scope of a handler for $\xi$ completely. The constructed tree will help us to determine these ranges.

### 8.4.2 Generating range tables

Generating the range tables, given the *try tree* that we have just constructed is fairly easy to implement. It is a nicely recursive process, which starts at the *root* and trickles down to the leaves of the structure, meanwhile splitting up intermediate ranges. We will briefly describe the algorithm for a strongly simplified type of tree. We assume, for example, that there is only one exception. Adding more exceptions does not really make implementation of the algorithm harder, it will merely make it harder to explain. We also assume there is only one try clause at the top level (i.e. it is not followed by sequential try blocks) and that this try contains a handler for the exception. The range table will then be created as follows:

1. Enter the range (try_begin, scope_stop, handler_label) in the table.
2. If there are nested try blocks, then for each sequential try clause that is contained, do the following:

   (a) If the try clause contains an exception handler, then split up that range table entry that contains the scope of the handler of this nested try. In other words, suppose the nested try has values \( \text{try \ begin} = \alpha \), \( \text{scope \ stop} = \beta \) and \( \text{handler \ label} = \xi \). Now consider an entry in the range table: \( \text{range}(r) = \langle a, b, c \rangle \). If the following holds: \( \langle a, b \rangle \supseteq \langle \alpha, \beta \rangle \), then replace \( \text{range}(r) \) by:
   - range \( \langle a, \alpha, c \rangle \)
   - range \( \langle \alpha, \beta, \xi \rangle \)
   - range \( \langle \beta, b, c \rangle \)

   (b) If the nested try clause contains nested try clauses itself then repeat (2) for these try clauses.

3. return the resulting range table

One last remark is needed with respect to generating range tables when there are multiple exceptions in the program. If we find handlers for \textit{finally} and \textit{default}, we should make sure that they split up the ranges in \textit{all} the other tables. This is because a \textit{default} or \textit{finally} at a lower level in the hierarchy subsumes all other handlers. Ordinary exception handlers on the other hand only split their own tables.

### 8.4.3 Translating exceptional C

Besides generating range tables, we also have to translate the exceptional C code to a program in which no more keywords like \textit{try} or \textit{except} occur. We will now address this translation in a very global way. Only a rough sketch is presented, restricted to what is absolutely essential to understand the implementation.

Before we start the description, we must remark that the implementation makes extensive use of \textit{labels}. These labels will be inserted in the program code, replacing the keywords \textit{except}, \textit{finally} and so on. In fact the ranges in the ranges tables will consist of these labels, to indicate which handler (\textit{label}) should be associated with the range denoted by \( \langle \text{try \ begin} (\text{label}), \text{scope \ stop} (\text{label}) \rangle \). These labels obviously need unique identifiers. For this purpose we maintain a variable \textit{label\_count} in the compiler. Each time a label is inserted, its identifier will be \textit{label\_count}, prefixed with \textit{\_EXNL}. The \textit{label\_count} is then increased by 1. This way, the uniqueness of the label identifiers is guaranteed.

Now consider the example of Figure 7.1. The exceptional part of this program (not including the \textit{raise} statement), is the following:

```c
try {
...
    except e1 (int arg1, int arg2, int arg3)
    {
        ...
    }
}
```

The translation of this program is illustrated in Figure 8.6. So we see that nothing really happens when we enter a try clause. The label does not incur any overhead. Even though
the Figure oversimplifies the real translation a little, the translation is not much different from the one shown. We also see that the code in the exception handler is simply skipped by jumping over it. Again, there is virtually no overhead in this operation. The only overhead that cannot be avoided, is incurred by finally handlers. At the end of a finally handler, we always have to test whether the finally was executed on behalf of a raised exception or not. If the execution was caused by an exception we have to make sure that this exception is reraised.

The labels are inserted in an intelligent way by explicit assembler instructions, because we do not really want the label names in our range tables, but the real addresses of where these labels reside in the code. Another option would have been to use the gcc extension that enables us to get to the address of labels. This solution is obviously a bit more straightforward. We have not chosen to use the extension because, for one, we were not able to make use of the labels outside the function. But even if it would be possible to do so, the solution would run into trouble when we execute the compiler with the optimisation option turned on, because normal labels, that are not jumped to will be deleted. We can prevent this from happening by explicitly inserting the labels using assembler statements (optimization will be discussed in detail in Chapter 9).

8.4.4 Implementing stack unwinding

Given the range tables, we still have to find a way to walk through the stack frames until we find the right handler for the raised exception (see Figure 8.7). For this purpose we will use an implementation that partly consists of (machine dependent) assembler code. On the DEC Alpha (and also on the MIPS architecture) this task is simplified because there is an administration structure, that keeps track of the operations that modify the registers. Examining this administration closely, enables us to restore the state of the registers. Our raise implementation is therefore:
Figure 8.7: Looking up the handler label in the range table for unwinding the stack

1. using the program counter, look up the address of the appropriate handler in the range table for this exception
2. unwind the stack until we find the place that corresponds to the label
3. construct a new \texttt{jmp_buf} in which we restore the context that corresponds to the context that existed at the point in the program hierarchy where the handler resides
4. execute a \texttt{longjmp} to restore the context
5. execute the handler

8.4.5 Range tables and output code in C

So far we have not paid any attention at all to the actual C code that must be generated. We have determined the ranges, but now we have to generate compilable C code for them. We will distinguish between what we call \textit{table code} and \textit{output code}. \textit{Table code} is the C code that is generated by the range table algorithm, it represents all the range tables for all the exceptions in the program. \textit{Output code} is the actual translation of the exceptional C code \textit{minus} these range tables. It contains the normal input code, in which the \texttt{“except”} keywords, for example, have been replaced by a label and some other statements. There are a number of things with respect to the output (both tables and translated code) that must be taken in consideration:

1. the table code should be completely \textit{static}, i.e. we should be able to just prefix the output code with it, without having to execute any code to initialise the tables,
2. the code should stay as close as possible to the types and modules that exist in \textit{Nemesis}.
The second requirement amounts to saying that we should use the Nemesis types and data structures, which is fairly easy. Most of the interfaces for dealing with exceptions are generated by the Nemesis interface definition language Middl and as long as we stick to the definitions in these header files, we will be fine.

The first requirement can be met, by generating code that is explicitly declared static and requires no computation. For example the code below, generated by XTC, can be prefixed to the output code and the handler table will simply be there when we start compiling:

```c
/* we first declare the labels we use as if they were functions */
static void _EXNLBL0();
static void _EXNLBL1();
static void _EXNLBL2();
static void _EXNLBL3();
static void _EXNLBL4();
static void _EXNLBL5();
static void _EXNLBL6();
static void _EXNLBL7();
static void _EXNLBL8();

/* here starts the actual handler table */
/* this particular handler table consists of three range tables: */
/* one for "exception1", one for "exception2" and one for */
/* "finally" */
static const ExnRegistry.Range _R0[1] = {
    &_EXNLBL6, &_EXNLBL7, &_EXNLBL7}; /* this is a range in the table */
static const ExnRegistry.Range _R1[1] = {
    &_EXNLBL0, &_EXNLBL1, &_EXNLBL1};
static const ExnRegistry.Range _R2[1] = {
    &_EXNLBL2, &_EXNLBL3, &_EXNLBL3};
static const ExnRegistry.ExnRanges _ER[3] = {
    "exception1", {1, (ExnRegistry.Range *) _R0}}, /* table for 1 exception */
    "finally", {1, (ExnRegistry.Range *) _R1},
    "exception2", {1, (ExnRegistry.Range *) _R2}};

/* _HT is the resulting handler table */
static const ExnRegistry.HandlerTbl _HT = {3, (ExnRegistry.ExnRanges *) _ER};
```

If we now combine the table code and the output code into a single file and compile this file together with the code that implements the stack unwinding, the result will be an executable program, that deals with exceptions in such a way that the user will hardly notice the mechanism, unless an exceptional condition is encountered. The generation and combination of files into a single program is illustrated in Figure 8.8.

### 8.4.6 Summary

All the components of fast XTC compiler have been discussed. We have seen the data structures that are built and the way they are built, as well as the general mechanism of dealing with exceptions. Context saving is no longer needed in this implementation, which results in an extremely low overhead in the normal case. Raising an exception, on the other hand,
will be slow. Range tables, generated at compile time, enable us to find the appropriate handler for a raised exception, given the program counter. A stack unwinding algorithm, finally, reconstructs the context that should exist when the handler is executed.
Chapter 9

Results and performance

Now that we have discussed all the relevant issues, having to do with the implementation, it is time we had a look at the results. Have we succeeded in implementing a flexible exception handling mechanism with reasonable performance? It is in that respect quite interesting to compare the two different implementations (portable and nonportable) to see how much performance we have gained with the low overhead design. Furthermore, both implementations should be set off against program execution without exception handling at all. For the portable solution we will consider both the setjmp and the nested function implementation. Performance will be measured on a DEC Alpha architecture.

9.1 Results

When we look at the two compiler implementations (i.e. the setjmp and the range table compiler), we should bear in mind that only one of these is a full grown compiler. It checks types and presence of exception handlers and produces corresponding error messages, when the input does not obey its (rather strict) rules for the syntax. The other implementation deals with a far more extensive syntax and translates the Exceptional C in a much more intelligent manner, but it is not a full grown compiler (yet). Also, only the first compiler succeeds in generating code that will run on any machine. For the other portable solution, using nested functions, we still haven’t implemented any compiler whatsoever.

We think that the syntax proposed in this thesis (and completely handled by the second implementation), is quite flexible, but still relatively easy to use (the proposal is based on [Evers, 1994]). It is an attempt to incorporate the best properties of existing exception handling mechanisms into a concise and straightforward language extension, without the disadvantages of the existing facilities. It even exceeds the mere combination of the good points of exception handling in existing languages, by introducing an implementation for the useful finally handler. As far as we know, this type of exception handling is not provided for in any of the other languages.

9.2 Performance

Whether the results are positive or negative, is not only determined by the issue “are exceptions handled elegantly?”, but also, by how the performance compares to program execution without the exception handling facilities. And indeed, even more so, because a mechanism,
no matter how elegant, that introduces an unacceptable penalty in the way of overhead, is in fact useless. Table 9.1 shows the performance results in the number of process cycles which were needed to execute the following loop:

```c
for (i = 0; i < loopsize; i++) {
    try {
        /* don't do anything */
        except (int argc, int *argv) {
            /* don't do anything */
        }
    }
}
```

So we measure the overhead introduced by the `try...except` only. After that, we will look at the performance when a function call is executed inside a `try` as well. To make our comparison between the three implementations as fair as possible, we use for the exception handler the type of argument format that the `setjmp` implementation requires (and which can also be handled by the nested-functions and range-tables designs).

The results are measured in process cycles, because we think this will give us the most accurate reading. Obtaining the number of process cycles requires some assembler code to correctly use the Alpha RPCC instruction (Read Process Cycle Counter [Sites, 1992]). In the measurement, we have left out all those cases in which the performance suddenly decreased, because of context switches. So, to make the result as reliable as possible, the program was executed a hundred times (by incorporating it inside a second `for` loop) for each loop size and for each implementation. For the performance we have taken the average number of process cycles that were found, leaving out all the extremely high values that were generated by context switches (although not shown in the table, we have in fact also left out very high values that may have been caused by cache misses).

Finally, for reference purposes, we have also entered a column containing the results for a completely empty loop. This allows us to compute the actual overhead introduced by the `try` statement. The overhead is obtained by subtracting the number of process cycles caused by an empty loop from the number of process cycles for the exception handling implementation. The reason for not doing the subtraction ourselves, but entering the results for an empty loop in the table instead, is that it allows us to consider the relative overhead caused by the different mechanisms as well (overhead measure in percentages is generally more usable than the overhead in process cycles).

If we compare the range table implementation of the exception handling mechanism with the empty loop, we notice there is hardly any performance difference between the two. The `try...except` seems to slow down the execution by a factor 1.06 approximately, which is hardly significant. So this implementation appears to make the grade. If we look at the performance of the `setjmp` implementation, however, it is ghastly. The low overhead implementation seems to be more than 100 times as fast as this portable exception handling mechanism. The nested function approach finds itself between these two extremes (although much closer to the range table design than to the `setjmp` implementation). It incurs about 3.5 times as much overhead as an empty loop and more than 3.3 times as much as the range table implementation.
Table 9.1: Performance results

<table>
<thead>
<tr>
<th>implementation</th>
<th>loop size</th>
<th>process cycles (on average)</th>
<th>context switches (per 100 loop executions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>setjmp</td>
<td>$10^2$</td>
<td>170585</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>$10^3$</td>
<td>1682193</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td>$10^4$</td>
<td>?</td>
<td>100</td>
</tr>
<tr>
<td>nested fn's</td>
<td>$10^2$</td>
<td>6288</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$10^3$</td>
<td>60513</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>$10^4$</td>
<td>620217</td>
<td>91</td>
</tr>
<tr>
<td>range tables</td>
<td>$10^2$</td>
<td>1879</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$10^3$</td>
<td>18088</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$10^4$</td>
<td>182938</td>
<td>11</td>
</tr>
<tr>
<td>empty loop</td>
<td>$10^2$</td>
<td>1676</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$10^3$</td>
<td>17308</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$10^4$</td>
<td>173196</td>
<td>9</td>
</tr>
</tbody>
</table>

These results clearly indicate that using `setjmp` to save contexts utterly destroys the goal of incurring very little overhead in the normal situation. It should be remarked, however, that these results are particularly bad, because they were obtained under Unix. The `setjmp` function in Unix is extremely slow, because not only all the registers are saved, but the signal state as well. Running the test under the Nemesis operating system, would probably be a bit more advantageous for this implementation. Furthermore, as long as `try...except` is kept outside loops, it generally does not matter that much, whether a `try` is slow, since it is only executed once anyway. Still, even given these two reservations, the stack unwinding implementation clearly outperforms the previous one.

**Nested functions versus range tables**

More interesting is the comparison of the nested function implementation with the range table compiler. How much faster is the range table design and is this gain in performance not outweighed by the disadvantage of the machine dependent code? This question cannot be answered easily. There are number of issues that have to be considered, each of which may be valued differently, depending on the particular preferences of the programmer.

So far, we have not dealt with most of these issues. Table 9.1 seems to indicate that the use of nested functions slows down the program by a little more than a factor of 3.3, compared to code in which the range tables are used. However, this only holds for normal exception handlers (and in case no exceptions are raised). Finally handlers are much more expensive, because they always involve the calling of the nested function and executing the code in it, even if no exception was raised. The range table implementation has no such additional overhead. On the other hand, the handling of raised exceptions in the nested function approach is much faster than in the range table code, because it requires little more than a function call.

Another serious problem with the nested function approach arises when, in a tight loop, we decide to **lexically nest** the try clauses. A nesting of depth 2 (which is quite common) means...
that, for each loop iteration, the overhead will be doubled, making the nested function about
6.5 times as expensive as code in which range tables are used. Adding another level of nesting
would make things even worse, but such nesting is rarely done by human programmers. Two
levels of nesting is much more common, because programmers frequently write code like:

```
try {
    ... get mutexes ...
    try {
        ...
        finally
        { ... release mutexes ... }
    }

    except normal_exception (args)
    { ... handle exception ... }
}
```

**Testing return codes**

Let’s consider the *low overhead* implementation and the *nested function* design, and their
observed overheads (less than 1.1 times and more than 3.5 times the delay caused by an
empty loop, respectively). Clearly, this comparison is not completely fair. To deal with
exceptional conditions, the program under consideration would certainly not run an empty
loop (or rather, a loop in which there is no overhead on account of exceptional conditions).
Instead, it would be constantly testing the return value(s) of the function(s) called. Such a
test costs at least one conditional statement. If we correct our results to take this effect into
account, we will find that the exception handling mechanism generates approximately the
same overhead as the conditional statement. And now we are only considering *one test for
one function*!

Table 9.2 gives the overhead incurred by doing a function call in the loop (see the test
program on page 64). We only consider those implementations that use nested functions or
range tables. The programs with exception handlers only call the function, without testing
its return value, but the conventional program (without exception handling) has to test this
value for each iteration. We do this in the standard way, which also requires an assignment
to effectively deal with the error, because this is generally needed and is the only method of
error handling that resembles exception handling:

```
if (!!((result = foo ()))) {
    handle_error (result);
}
```

The overhead for the function call is constant (approximately 20 cycles for each call). It
follows from the table that even for *one* function call the performance of the *range table*
implementation is as efficient as testing the return value. This means that there is no penalty
whatsoever for using the exception handling mechanism. Remember, that the test in the
conventional program was also one of the cheapest tests possible. The *nested function* approach
is still a bit slower, but if more functions were called, it would rapidly overtake the
return-code program, because the overhead for the function call would be incurred in both
programs, but the code with exceptions has no need for explicit testing of the return value.
### 9.2. PERFORMANCE

<table>
<thead>
<tr>
<th>implementation</th>
<th>loop size</th>
<th>process cycles (on average)</th>
<th>context switches (per 100 loop executions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>nested fn's</td>
<td>$10^2$</td>
<td>8381</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$10^3$</td>
<td>81068</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>$10^4$</td>
<td>801402</td>
<td>77</td>
</tr>
<tr>
<td>range tables</td>
<td>$10^2$</td>
<td>3750</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$10^3$</td>
<td>35221</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>$10^4$</td>
<td>356168</td>
<td>19</td>
</tr>
<tr>
<td>return code</td>
<td>$10^2$</td>
<td>3623</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$10^3$</td>
<td>35620</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>$10^4$</td>
<td>355156</td>
<td>9</td>
</tr>
</tbody>
</table>

#### Finally handlers and context switches

Another issue, that must be taken into consideration has to do with finally handlers. In all implementations, a finally handler is slower than a normal exception, because it must always test whether the exception has to be raised. This is also amounts to exactly one conditional statement. We have shown already that finally handlers in the nested function implementation are particularly slow, because the handler function must always be called (probably inlined), and its code executed.

We also observe that for every test program that we have executed, the number of context switches for the *nested function* solution is much higher than the number of context switches for the *range table* design (see Figure 9.1). These context switches introduce a huge amount of overhead. We have not included the resulting performance in our tables, because the overhead is generally not introduced by the mechanism itself. We should remember, however, that this is of no concern to the users of the mechanism. They only observe that program execution is very slow. In that respect, the *nested function* approach suffers from this additional overhead of context switches more often than the *range table* solution.

#### Raising exceptions

The overhead incurred by try clauses is of much more importance than overhead that accompanies the raising of an exception. Nevertheless, it is interesting to see how much slower our *low overhead* exceptions are, when compared with the *setjmp* exceptions. Unfortunately, we have no reliable performance results for the *nested functions* approach, because we have not implemented a good way to pack and unpack the arguments that are passed to the handler for it (because the rest of the overhead only consists of a function call and a *switch*, this is likely to constitute the bulk of the overhead). It should be clear, however, that packing and unpacking of arguments is done in the other implementations as well, so that the *raise* statement in the nested function design can be expected to be much faster than the *raise* statements in the other two implementations (we have tested useses without argument passing and the number of process cycles needed for doing just the basic things turned out to be no more than 325).
Figure 9.1: Number of process cycles for 100 executions of an empty loop of $10^4$ iterations

To test the overhead created by the raises, we measure the process cycle counter right before execution of the raise statement and immediately after execution of a completely empty handler. Table 9.3 shows the results in process cycles. The results should be taken as rough indications, because we have not executed the raise statements in long loops. We have, however, executed them a thousand times, and taken the average of these computations, leaving out cache misses and context switches. We see that the setjmp implementation is approximately four times as fast as the stack unwinder.

<table>
<thead>
<tr>
<th>implementation</th>
<th>process cycles for raise statement (average of 1000 executions)</th>
<th>switches and misses (per 1000 executions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>setjmp</td>
<td>4534</td>
<td>37</td>
</tr>
<tr>
<td>range tables</td>
<td>16600</td>
<td>19</td>
</tr>
</tbody>
</table>

Exceptional C: design and implementation of the XTC compiler
9.3 Optimisation

9.3.1 The optimisation problem

We always want our code to run as fast as possible, but particularly so in an environment where operating systems are being developed. This is because many things in such an environment are very time critical. For this reason, we generally run our code through a so called optimiser. An optimiser is generally part of the compiler and it removes code that cannot be reached, reorders statements, when it finds this will increase speed (and doesn’t alter the program behaviour), and so on. For an exception handling mechanism to be generally applicable, we must make sure that it intelligently deals with optimisation options. However, this generates some tricky problems.

From now on, we will just consider our fast implementation, partly, because we feel that to be the most promising approach, but more importantly, because this is the design that encounters more problems when optimising than the other two implementations. This is because the two portable implementations only use standard gcc code, which the compiler can handle (for example, the removal of code that appears dead, which will be discussed next, can never occur in these implementations, because the compiler is aware of the fact that all the code is reachable). The range table implementation, however, uses all sorts of tricks, that are intrinsically foreign to the compiler, so that the optimisation strategy may be based on assumptions that are no longer valid. Where appropriate, we will explicitly indicate issues that are of relevance to the other implementations as well.

Now consider figure 9.2. Here we see, once again, our translation of a try...except block (leaving out unnecessary details). A jump is always made over the except block and there will never be a jump back to the label of except, by way of a goto. There is no reason for the compiler to assume that this code is not dead. So the first thing that the optimiser does, is delete this block of code. In reality, however, the code is not unreachable at all! Such an optimisation would ruin the whole exception handling mechanism.

```c
try {
    ...  
    goto endtry;
    //except (int arg) {
    //    ...  
    //}  
    endtry:  
    /* endtry actions*/
}
```

Figure 9.2: Compiler optimisation: dead code
Reordering code

It gets even worse, because the results of reordering code inside a try may also be quite disastrous. The optimiser sometimes takes a piece of code, and migrates it to another position in the program, if it feels this will not affect the results. For example, if the compiler would come accross (valid) code like:

Exceptional C code:

```c
try {
    int x = 0;← initialised at all times
    call_fn(); /* and call_fn() can raise xption1 */
    x = whatever;
    except xption1 () {
        if (x) do_something();
    }
}
```

it may decide to optimise this by translating it to any of the following (invalid) programs:

Translation 1:

```c
try {
    int x;← not initialised, because flow detects no use; this is invalid exceptional C, however, because the exception may be raised while x is still uninitialised
    call_fn(); /* and call_fn() can raise xption1 */
    x = whatever;
    except xption1 () {
        if (x) do_something();
    }
}
```

Translation 2:

```c
try {
    int x = whatever;← initialised at all times, but wrong value
    call_fn(); /* and call_fn() can raise xption1 */
    except xption1 () {
        if (x) do_something();
    }
}
```

It will do this translation, if the latter piece of code is more optimal than the original. As far as the compiler can tell, changing the order in this way, does not effect the external behaviour of the program. In reality, however, it does effect the behaviour, if call_fn () decides to raise exception “xption1”.

The optimiser may even decide to translate the code to the following program (“translation 3”). In this translation, the compiler removed the assignment for x altogether. The reason for doing this is that flow analysis seems to indicate that x is never used, so there is

---

Exceptional C: design and implementation of the XTC compiler
no need to initialise at all:

Translation 3:

```c
try {
    int x;← x is apparently never used, so don't initialise
    call_fn(); /* and call_fn() can raise xption1 */
    except xption1 () {
        if (x) do_something();
    }
}
```

All of these translations lead to invalid program code, caused by the compiler's inability to correctly deal with the new flow control that comes with exception handling.

### 9.3.2 Solving the optimisation problem

To prevent the optimiser from clobbering our try blocks, we have to resort to using assembler code again. In our implementation we have dealt with quite a number of optimisations, but it is doubtful whether we will ever be able to deal with all of them, unless we fully integrate the exception handling in the language itself, i.e. from the ground up. When we extend a language with a new control structure, unknown really, to the internals of our compiler, we will find that the whole optimisation scheme is built on assumptions that no longer hold.

For example, the optimisation we demonstrated last, is based on the assumption that we can just move code around in a try block, because, internally, the compiler will have no notion of what a try clause with exception handlers is. It is a concept we have enforced upon the compiler. This works fine as long we can keep matters in our own hands, control what code is placed at what address and so on. As soon as we let the compiler mess around with these things, based on assumptions that are no longer valid, things will start to go wrong. So all we can do is to try and make it work as good as possible, even when optimisation is turned on.

The solution for each of the problems uses some specific knowledge about how the compiler works internally, and uses code that will explicitly circumvent that particular optimisation. As an example of how this is done, we will discuss the case of the code that appears dead, and a type of optimisation that can be solved in similar fashion. A more general solution in which the exceptional statements are introduced at a low level and integrated in gcc itself (and which has not been implemented yet), will be discussed in paragraph 9.3.3.

### Saving apparently dead code

Preventing blocks of code from being declared unreachable is done by starting each block with a label with local scope. The compiler will delete such labels if it can be sure that no code will ever branch to it. We can prevent this by making an assembler statement in the same block appear to do that. Since gcc (the compiler we use) can’t parse the contents of the assembler code, it can’t be certain that the block can never be reached, so it is never deleted. We pass the label to the assembler code by giving it as an input parameter to the statement with: "\$x" (\$label\$name). This has no code overhead, since we refer to the memory address, but never actually need an instruction to load it.

---

Exceptional C: design and implementation of the XTC compiler
Duplicate code elimination

Another optimisation problem that can be solved in a similar fashion, deals with *elimination of duplicate code*. This is the optimisation that detects that two blocks of code are the same and decides to create only one binary (one piece of assembler code) for them. If two try blocks are exactly the same, it may therefore decide to map them on one block of code. The result of such an action would destroy the validity of our range tables, because they were based on the assumption that we can find the appropriate handler (which is generally a different one for each of the try clauses), by just looking at the program counter, when an exception is raised. The program counters for the two try blocks, however, are in this case exactly the same (they are executing the same block of code). So, the stack unwinding mechanism is no longer able to determine the right exception handler in all cases.

We must therefore make sure that this elimination will not take place. This can be done by surrounding each block of code (i.e. try blocks and handlers) with a local label at the start of the code and a reference to this label at the end of the code. In this way each piece of code references a distinct label and it will not be eliminated by the optimiser.

Code migration

The other problem, code migration in try clauses, is much harder. We have succeeded in finding a solution, that prevents the optimiser from moving code across labels (e.g. out of the exception handler and into the normal try code). In the example shown, however, the migration takes place inside the try block itself. For this reason we use the following rule for code that needs to be optimised: any variable that is altered in a try clause or referenced in an exception handler, must be qualified volatile. Note that specifying these variables to be volatile is not only necessary in the range table implementation. The same problems arise in the two non-portable solutions, so here we have to declare the variables in the same way.

The purpose of the ANSI C type qualifier volatile is to force an implementation to suppress this sort of optimisation [Kernighan and Ritchie, 1988]. If we specify the variables to be volatile all the bad translations of 9.3.1 will no longer arise. In fact, the setjmp implementation of the exception handling mechanism will generate warnings, if variables are in danger of being clobbered, because they are not specified as volatile.

Other forms of optimisation

The *gcc optimiser* has more optimisation strategies than the ones described here. We have dealt with a number of them, of which we feel they are the most important. It is essential, however, that our compiler will one day be able to cope with all types of optimisation. For this purpose (and for integrating the whole mechanism in the compiler proper), some future research is needed. Although we are confident that the mechanism as it is now, will work fine in almost all cases, we must, until then, take caution when we switch optimisation on.

9.3.3 General solution for the optimisation problem

The approaches to solving the optimisation problems discussed so far, aim at removing the hazardous effects of particular optimisation strategies. Such solutions boil down to fighting symptoms, instead of attacking the cause of the problems, which is that the compiler has no real notion of the new language constructs. A more general solution would set about
introducing these concepts to the internals of the C compiler. This task involves some low
level work in gcc, in particular at the level of the register transfer language (also known as
RTL). In this paragraph we will sketch how such a task can be accomplished.

**RTL: register transfer language**

In gcc most of the work of the compiler is done on an intermediate representation called register transfer language. By “intermediate representation” we mean that the C statements are first translated to RTL, before the translation to real machine code is executed. At this level the compiler will do most of the processing on the code. It is the level, for example, where all optimisation takes place. In this language, the instructions to be output are described, more or less one by one, in an algebraic form that describes what the instruction does. For example, an assignment like:

\[ x = a; \] will be translated to: (insn (set (reg x) (reg a)))

**Issues**

When we have a closer look at the problems that are caused by optimisation, it can be observed that the issue of apparently dead code in the exception handler can be solved by inserting a reference to its label. At the moment, this is done by an explicit assembler statement inside the handler itself, but the place of reference is obviously not very important. The ideal solution no longer needs an assembler statement to indicate that the code is reachable: the reference will be integrated in the RTL code. How this is done will be discussed in the next paragraph.

The introduction of the concepts of **try...except** to the language would also have to mark the **try block** in such a way that duplicate code (for two try blocks with different labels) is not eliminated.

The last problem has to do with moving code across potential activation points of exceptions. A potential activation point of an exception is, for instance, a raise statement, or a call to a function that may raise that particular exception. If the handler code is in any way dependent on the migrated code, erroneous program behaviour is imminent. Note that the migration of code is **no** problem if it does not involve moving the statements across such an activation point. For example, consider the following code:

```c
[1] try {
[2] \text{x = a;}
[3] \text{y = b;}
[4] \text{foo (args); /* "foo" raises "e1" and "e2" */}
[5] \text{x = c;}
[6] \text{except e1 (int argument)}
[7] \{  
[8] \hspace{2em} \text{if (x) do\_something ();}
[9] \}
[10] \text{except e2 (int argument)}
[12] \hspace{2em} \text{if (y) do\_something\_else ();}
[13] \}
[14] }
```

---

**Exceptional C: design and implementation of the XTC compiler**
The potential activation point in this program is line 4. The only place where an exception can be raised is in the function call. This means that it does not matter, if the order of the statements in lines 2 and 3 is changed, or even if these two assignments are migrated out of the try block! This will not affect the program behaviour in any way. Moving statements (e.g. assignment 5) across the function call in line 4, however, may lead to erroneous behaviour.

Our solution should take this factor into account. For example, a rather obvious solution would be to protect every single statement in the try block by surrounding it with a label declaration and a reference to this label. This would prevent gcc from migrating the code. It is, however, an over-aggressive strategy, because most migrations are perfectly safe. What we want is to allow optimisation to process all of the code, not just code outside try blocks. It only has to recognise and deal with the specific characteristics of exception statements appropriately.

For example, suppose we have a try block containing a raise statement and the corresponding handler is associated with this try itself. In that case, the raise statement is more or less the same as a goto statement, jumping to this handler. So, if we translate the raise in RTL like a goto, optimisation will present no problems: it can deal with goto statements, so no code migration that may change program behaviour, will occur.

Similarly, if the exception is not raised by an explicit raise, but by a function call (remember we can check which exceptions a function can raise apriori), we must insert RTL code, indicating that “control may be transferred to the exception handler, when executing this function, instead of returning normally.” This means that the referencing, that was done by an assembler statement, will be inserted in the RTL code. Thus, the assembler reference will be rendered superfluous and may be deleted. This has the additional advantage, that all normal optimisation can take place. For example, if we find no raise statement and no function call that may raise a certain exception, there will be no reference to the exception label and therefore, the handler will be deleted. This is correct, because it is, in fact, dead code.

Summarising, the following has to be taken into consideration:

- exception concepts must be introduced to the language itself, at the RTL level,
- all optimisation should be allowed, even inside the try block (it should just handle the exceptional code correctly),
- code migration is a problem only when the code is moved across potential activation points,
- raising an exception using the raise statement is very similar to executing a goto (especially if the handler is available in this try); translating the raise to a jump will therefore allow the optimiser to work normally,
- function calls will have to indicate that they may transfer control to the exception handler, instead of returning normally.

The solution

It should be clear by now, what the issues are, when implementing a solution for exception handling, that can cope with all possible forms of optimisation. In appendix B this ideal solution in RTL for the previous example is illustrated. We will not discuss this solution at

---

Exceptional C: design and implementation of the XTC compiler
9.3. **OPTIMISATION**

great length in this paragraph, but there are some points that deserve a little more attention. The solution works fine with optimisation, because it has really integrated the exception handling mechanism in the compiler and the RTL code.

Furthermore, it demonstrates the power of the syntax we proposed in chapter 7 and particularly of the explicit declaration of exceptions that may be raised by a specific function. This is because the last point in the previous paragraph stated that “function calls will have to indicate that they may transfer control to the exception handler, instead of returning normally” and with the proposed syntax, the compiler can check which exceptions may be raised by this function and, therefore, to which handlers control may be transferred! This allows much better optimisation than would be possible if it only knew that a function call might raise exceptions, but not which ones (i.e. if there is no explicit declaration). In the latter case, a function call would have to indicate that control may be transferred to *all* handlers associated with this *try block* to prevent elimination of code that appears dead and this would have to be done for *each* function call in the try (even the ones that will never raise anything), because each function may potentially raise exceptions.

Worse still, less extensive code migration for optimisation *inside* the try statements themselves would be possible. For example, suppose a *try block* has exceptions $\varepsilon$ and $\xi$ associated with it and contains a function call $\mathcal{F}$ that can only raise $\xi$. If the handler for $\xi$ is in no way dependent on a variable $x$, while the handler for $\varepsilon$ is dependent on this variable, we can observe the following:

- In a solution that uses explicit declaration of contagious exceptions, the compiler is free to move code that changes $x$ across $\mathcal{F}$, because it *knows* control can only be transferred to handler $\xi$, which is totally independent of $x$.

- A solution *without* this explicit declaration, however, is never sure which handler may be executed, so it cannot permit this code migration to occur, even if it would make the resulting code much faster.

The result is that our proposal has some advantages over other (proposed) implementations. In particular, the GNU cooperation is working on implementing exception handling facilities without explicit declaration in a future release of gcc. It will be difficult in such a syntax to obtain the same level of optimisation as would be possible in a language that prescribes explicit declaration of raisable exceptions.

### 9.3.4 Performance of optimised code

Finally, we will briefly discuss the results with respect to performance when we switch the optimisation options for the compiler on. Only *low overhead* exception handling and exception handling based on nested functions will be considered, because these are the only implementations that offer an acceptable performance. Table 9.4 shows the results in process cycles for the optimised code. We see that the gap between exception handling and “nothing at all” has grown slightly wider, but not much so. All performances seem to improve considerably (by more than a factor 2 for the two exception handling programs and close to a factor 4 for the empty loop, respectively). If we bring to attention again, that an empty loop is not at all realistic, we can only be pleased with these results.
<table>
<thead>
<tr>
<th>implementation</th>
<th>loop size</th>
<th>process cycles (average)</th>
<th>context switches (per 100 loop executions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>nested fn’s</td>
<td>$10^2$</td>
<td>3748</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$10^4$</td>
<td>37944</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>$10^8$</td>
<td>375677</td>
<td>42</td>
</tr>
<tr>
<td>range table</td>
<td>$10^2$</td>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$10^4$</td>
<td>9067</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$10^8$</td>
<td>91900</td>
<td>4</td>
</tr>
<tr>
<td>empty loop</td>
<td>$10^2$</td>
<td>4043</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$10^4$</td>
<td>40750</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 9.4: Performance results for optimised code
Chapter 10

Conclusions

We have described three implementations for exception handling in C, two of which were built as (pre)compilers. The first approach attempts to arrive at a portable implementation of this mechanism, using the *setjmp* library routine, the second by using *nested functions*, while the third is completely machine dependent, using assembler routines that only work on DEC Alpha workstations. Although the design and implementation of exception handling is aimed at providing these facilities to the *Nemesis* kernel, they are not restricted to this system. In fact, the test results were all obtained while running the exceptional code under *Unix*. We have discussed the respective performances, both with and without raising exceptions. Furthermore, for the nonportable implementation, we have looked at the problem of compiling with optimisation and discussed various ways to deal with it. It is now time to evaluate the mechanism and draw some conclusions.

10.1 Evaluation

First of all, we should determine which goals have and which ones have not been reached. Second, we will discuss what interesting possibilities there are for future research.

We think that, in general, the implementation of the exception handling has been quite successful. We have demonstrated that exception handling in C is feasible and not only that, it is a rather powerful extension as well. Note that there are only two implementations of the XTC compiler, one using *setjmp* and another using *range tables*. None of the requirements, that were set back in chapter 5, seem unachievable and most were reached completely. All things considered, we don’t believe that the implementation using *setjmp* is the way to go. No matter how portable, it is just too slow to be of any practical use (except for experimenting maybe). The overhead it generates in a loop is about 150 as much as the overhead of an empty loop. So the conclusion has to be that the only implementations that are eligible for large scale programming are the one that uses *nested functions* (although we still have to implement a proper compiler for it) and the one that uses genuine *stack unwinding*.

It is hard to choose between these two, because they both have some very desirable properties. The portability of the former approach outweighs many of the advantages of the range table implementation. On the other hand if we emphasise the requirement of *low overhead*, the range tables are clearly the way to go, because they result in code that has virtually no consequences at *run time*. Furthermore, we have sketched a future implementation of the range table implementation that removes the problems of optimisation as well. As for the
portability, we should try and design stack unwinding algorithms for other architectures as well (a preliminary version for the MIPS processor exists already).

Our conclusion is that, unless we really need the speed of the machine-dependent solution, the nested function design is probably more usable, for the time being. It is not the ideal solution, but it certainly beats the setjmp code. On the long run, the range tables look more promising. The nested functions always incur unnecessary overhead, that can be avoided by using range tables. The ideal solution, we think, is a range table implementation, in which the optimisation problems have been solved in the way indicated in paragraph 9.3.3 and which is completely integrated in gcc (with implementations on as many different architectures as possible). So, on a short term basis, we choose the nested functions, but on the long term, the range table solution is more desirable.

If we base our evaluation on the latter (after all, there is not even an implementation of XTC with nested functions), we see that we are able to deal with a versatile, very C-like syntax, but we also observe that there is no full grown compiler yet. This means that a lot of type checking and error generation is not possible in this version. Even so, the results are very promising and, if we ignore the optimisation problem for a moment, it should not be very difficult to incorporate the whole mechanism into gcc itself.

We highlight some of the requirements mentioned in chapter 5, in a little more detail:

- **Hierarchical scopes and nesting**: this goal was achieved completely. The exception handling is based on a hierarchy that can be specified both by explicit (lexical) nesting and by calling functions that have their own try..except structure.

- **Normal, default and finally handlers**: all of these are provided in the exception handling facilities. Especially the finally handler proves to be a useful addition to the exception handling facilities in existing languages.

- **Argument passing**: argument passing is made as general as possible. There is no type checking on the arguments, however, because the mechanism is not yet an integral part of gcc and, for the same reason, we are not yet able to deal with arguments in raise statements, as if they were real expressions. So, for example, we cannot pass a function call to an exception handler, even if the result is of a type the handler can deal with. Also, expressions using brackets, operators, or in general, requiring computation, are not supported. This is not inherent to the mechanism, however, they could be easily added, when we move the mechanism inside gcc.

- **Low overhead in normal situations**: as opposed to the setjmp implementation of XTC, we find that the stack unwinding version results in very low overhead indeed. It even deals with a number (but not all) optimisation strategies gcc employs.

- **Static association based on the termination model**: these two goals have been reached completely. The former by associating handlers with the try blocks that contain them and the former by simply not supporting a resume operation.

We see that there are a number of minor points (such as expression evaluation in argument passing, optimisation, etc) on which the xtc compiler needs some improvement. On the whole, however, we have succeeded in building a (pre)compiler, that we think able to deal with exceptional conditions, in a way that guarantees a more reliable code with the same performance as the original program (or even better: if there are multiple tests of return
codes, at different levels of abstractions this is bound to be slower than the overhead incurred by the *try block*).

It should be remarked, furthermore, that implementing a complete compiler that thoroughly deals with exception handling (i.e. with all optimisation strategies, type checking) for a whole range of machines, requires infinitely more time than we were able to spend. Given this limitation, we think that the resulting XTC compiler is about as good as we could possibly get it.

### 10.2 Future work

There are quite a number of things that are left to be done. First of all, we should incorporate the whole mechanism into the *gcc* compiler proper. This step is essential if we want the arguments in raises to be normal expressions, do type checking for them and check the correct use of exceptions in general. From there, we should thoroughly examine *gcc*’s optimisation strategies and adapt them for dealing with try blocks, exception handlers, etc., instead of adapting the mechanism to deal with the optimiser, which is what we have been doing so far. The solution for the optimisation is sketched in paragraph 9.3.3. It requires generating some extra *RTL* code and is probably not the easiest of tasks.

As an intermediate step we might consider making the exception handling facilities suitable for working with multiple threads. This will be relatively simple. For this purpose, we should provide each thread with its own exception handling stack and maybe add some global data to control the whole lot. This extension is very straightforward, but quite useful nevertheless. It should also allow one thread to raise exceptions in another thread, enabling us to exercise more structured control over exceptional conditions in a parallel fashion.

Another easy step is to make the exception handling mechanism really work with Nemesis. There should be no problems there, it just means that we will have to use the Nemesis way of calling functions and generate and use Nemesis interface specifications in Middl and so on.

When all these recommendations have been implemented we will have an exception handling mechanism, that enables programmers to make their code more reliable in a straightforward manner without having to sacrifice anything in terms of performance. Furthermore, we will have a control structure that cannot be ignored by forgetful programmers. No matter what happens at intermediate levels, as long as we provide the right handler at some high level in the hierarchy, exceptional conditions, such as errors, will no longer cause erroneous behaviour of the software (there is no need to check results at intermediate levels). Finally, the normal and abnormal cases will be clearly separated, which has the additional advantage that the mechanism will clarify the program specifications.
Bibliography


Appendix A

A complex example

The following source code will generate the try tree shown in figure 8.2. Try clauses are nested in a normal way up to three levels deep. An additional nesting level is obtained by putting a try clause in a handler as well (this is the try...finally in exception “xption_e1” of try 1.1.1). Note that the code is not unique. Any code, having the following exception structure would create the same try tree. Note also, that in general using this many levels of nesting is not very useful and I think programmers are not very likely to come up with more than two levels of nesting.

    /* first a very complex try block with a lot of nesting */
    try {
        printf("statement in try 1\n");
        try {
            printf("statement in try 1.1\n");
            try {
                printf("statement in try 1.1.1\n");
                except xption_e1 (int args) {
                    printf ("handler for ‘xption_e1’ in try 1.1.1\n");
                }
            }
        }
    }
    except xption_e2 (char *args) {
        printf ("handler for ‘xption_e2’ in try 1.1.1\n");
    }
    except default {
        printf ("default handler for try 1.1\n");
    }
    try {
        printf("statement in try 1.2\n");
        except xption_e2 (int args) {
            printf ("handler for ‘xption_e2’ in try 1.2\n");
        }
    }
    except xption_e1 (char *args) {
        printf ("handler for ‘xption_e1’ in try 1.2\n");
    }
    except default {
        printf ("default handler for try 1.2\n");
    }

73
printf ("handler for 'xption_e2' in try 1.2\n");
}
}
extcept xption_e1 (int arg) {
    printf ("handler for 'xption_e1' in try 1.1\n");
}

/* now a sequential try at top level */
try {
    printf ("statement in try 2\n");
    finally {
        printf ("finally handler\n");
    }
}
Appendix B

Ideal implementation in RTL

We will now discuss what the ideal future implementation of exception handling would look like at the RTL level. This implementation has not yet been realised in gcc and I think it will take quite some time to efficiently do so. Nevertheless, I believe it to be a welcome and even essential job, if the exception handling mechanism is to be generally applicable. Consider the example on page 73. The translation in RTL for this piece of code would ideally be something like:

```
[1] (note note_try_top)
[2] (code_label range0_begin)
[3] (insn (set (reg x) (reg a)))
[4] (insn (set (reg y) (reg b)))
[5] (insn (set (reg parm0) (real_parm0)))
[6] (insn (set (reg parm1) (real_parm1)))
[7] (call_insn (call (label (foo))))
[8] (uses (reg parm0) (reg parm1)...)  
[9] (clobbers (reg clob0) (reg clob1)...)  
[10] (raises (label (except_lab0)) (label (except_lab1))...) / 
[11] (insn (set (reg x) (reg c)))
[12] (note note_try_end)
[13] (code_label range0_end)
[14] (jump_insn (set (pc) (label (end_try))))
[15] (barrier)
```
Although we do not mean to give an extensive explanation of RTL, we will highlight some relevant issues in the translation. We will do this by referring to its line number and briefly explaining its meaning.

**Line Description**

1. *Notes* are generally used to represent additional debugging and declarative information. In this case it indicates that a try clause is about to begin (and for two such clauses that are the same, we must never eliminate the duplicate code).

2. This is the "try_begin" label.

3-4. The two assignments \( x = a; \) and \( y = b; \)

5-10. The function call to \( \text{foo} \) (I assumed \( \text{foo} \) was called with two parameters). The *uses* indicates the registers (other than the stack pointer), not explicitly mentioned in its RTL, that this function requires. Similarly, the *clobbers* expression indicates which registers are destroyed. Both of these expressions exist already in RTL. The *raises* expression, however, is new. It will have to be introduced to RTL and obviously indicates which exceptions may be raised by this function by referencing their labels. This will make the handler code alive at the same time.

11. The assignment \( x = c; \)

12-13. Here we find the note and the label for the *scope_stop*.

14. The *goto endtry* of our translation.

15. Barriers are placed in the instruction stream when control cannot flow past them. They are placed after unconditional jump instructions to indicate that the jumps are unconditional.

16. This is one of the labels mentioned in the *raises* expression. It marks the beginning of the first handler (the handler for "el").

16-20. All rather obvious. No translation is given for the conditional statements in the handlers.

21. This is the *end_try* mark, implemented by a label.
We have not extensively discussed all the RTL instructions and we have ignored some very important issues as well (like what happens when the handler is not found in this try clause), but this translation is what the exception handling in RTL for optimisation purposes boils down to. Most important, is of course the \texttt{raises} expression in the function call. If this is implemented, we will have a powerful exception handling mechanism (namely the one discussed in this thesis) that combines very well with all possible optimisation strategies. As is shown in paragraph 9.3.3, the optimisation will be even more powerful than many other implementations, because of the explicit declaration of contagious exceptions.
Appendix C

Example range table program

Consider the following program in exceptional C:

```c
char tmp[80] = "Global string will be used as argument\n";
int Globi = 100;

typedef struct ExArgs { char s[80]; int i; } Args;

static void
inner_blow_up (unsigned int i) raises (e1, e2)
{
  int *ip = &Globi;
  Args arg1;

  strcpy (arg1.s, tmp);
  arg1.i = 10;

  raise exception1 (arg1, "this is a const string arg", i, ip);
  printf ( "--- oops! after RAISE in inner_blow_up\n");
}

static void
blow_up (unsigned int i)
{
  try {
    printf ( "\tbefore inner_blow_up (i=%d)\n", i);
    inner_blow_up (i);
    printf ( "\tooops! after inner_blow_up\n");
    finally
    {
      printf ( "\tin inner finally %d\n", i);
      try {
        printf ("\tt\tt in finally\n");
        raise e1 ("test arg"); /* raising with constant */
        except e1 (char *error)
        {
          printf ("\tt\tcought args: %s\n", error);
        }
      }
    }
  }
}
main (int argc, char *argv[])
{
  printf ("test starts\n");
  printf ( "try_test: Main: argc=%d argv[0]=%s\n", argc, argv[0]);

  try {
    printf ("try stmt\n");
  }
}
```

blow_up(10);

/* we show exception handler can handle many argument types */
except exception1 (args arg1, char *arg2, int arg3, int **last)
{
    int i;
    printf ("Caught exception 1 with args:n");
    printf ("arg1.s = %sarg1.i = %d\narg3 = %d\nlast = %d\n", 
        arg1.s, arg1.i, arg2, arg3, **last);
}
}

This program will be translated to the following. To make the code readable, we first give an include file, in which the macros are defined. In the next file, we find all the range tables and global variables, etc. The include file looks like:

/*
* raise_impl.h (Experimental Version)
* exception handling macros - without setjmp, using GNU C extensions
*/

/* Note: many anti-optimisation code has been entered in this file. For
* the labels "_exception_opt1" and the references to it
* ("_asm_ volatile (" : : "X" (__asm_ _exception_opt1));") are only there to
* prevent elimination of apparently dead code and code that is duplicate
* (but needs to remain a distinct region and cannot be folded into a single
* piece of code)
* E. Bos (12-9-1994)
*/

/* A TRY block is a GNU C compound statement, in order to allow use
* of local labels.
*/

#define _TRY_h_
#define _TRY_h_
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "ExnRegistry.h"
#include "Stack.h"

typedef unsigned long t_addr;

/* copied from ranger.h */
#define ALL "all"
#define FINAL "finally"
#define TRY(id) \
{
    _label__ _entry;
    __asm_ _volatile (".asent _EXNLBL" #id "; _EXNLBL" #id "; ; : : 
        "$0", "$1", "$2", "$3", "$4", "$5", "$6", "$7", "$8", "$9", 
    ((_label__ _exception_opt1; _exception_opt1: { 

#define CATCH(id, e) \
        goto endwhile;

________________________________________

Exceptional C: design and implementation of the XTC compiler
The previous code defined the macros that are used in the following translation of the program:

/* start of range table code generated by ranger */

#include "includefile.h" /* this is the previous include file with macro definitions */
static void _EXNILIO();
static void _EXNILII();
static void _EXNILII();
static void _EXNILII();

---

Exceptional C: design and implementation of the XTC compiler
static void _EXNLBL4();
static void _EXNLBL5();
static void _EXNLBL6();
static void _EXNLBL7();
static void _EXNLBL8();
static const ExnRegistry_Range _R0[1] = {
    {&_EXNLBL6, &_EXNLBL7, &_EXNLBL7});
static const ExnRegistry_Range _R1[1] = {
    {&_EXNLBL0, &_EXNLBL1, &_EXNLBL1});
static const ExnRegistry_Range _R2[1] = {
    {&_EXNLBL2, &_EXNLBL3, &_EXNLBL3}};
static const ExnRegistry_Errs _ER[3] = {
    {"exception", {1, (ExnRegistry_Range *) _R0}},
    {"finally", {1, (ExnRegistry_Range *) _R1}},
    {"e1", {1, (ExnRegistry_Range *) _R2})};
static const ExnRegistry_HandlerTbl _HT = {3, (ExnRegistry_Errs *) _ER};

/* this should really be a dynamic stack */
typedef char _xption_name_[30];
_xption_name_ _EXN_LAST = "";
_xption_name_ _EXN_NAME_ [10] = {'",","","","","","","","",""'};
int _EXN_NAME_ = 0;
int _EXN_ARGS [10] = {NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL};
int _EXN_ARG_LAST = 0;
int _FROMFINAL = 0;
int _DOMAISE = 0;

/* end of ranger output */
/* the global for communication between "raise" function and finally's */

int _FROMMAISE = 0;
/* Implementation of the "raise" function for exception handling */
/* The range tables are assumed to be generated already by the "ranger" */
/* and should be assigned to "_HT" (i.e the handler table). */

ExnRegistry_CodePtr
ExnRegistry_CodePtr pc, ExnRegistry_Ranges *XR RR
{
    int i;
    for (i = 0; i < XR RR->len; i++)
        if ((XR RR->data)[i].start <= pc && pc < (XR RR->data)[i].stop)
            return (XR RR->data)[i].handler;

    return (ExnRegistry_CodePtr) 0;
}

void _raise (char *id, int *args)
{
    ExnRegistry_HandlerTbl *XR HT = &HT;
    Stack Frame f;
    int i, x_index = -1, all_index = -1, final_index = -1;
    ExnRegistry_CodePtr handler, hand2;

    /* find appropriate range table */
    for (i = 0; i < XR HT->len; i++)
        if (!strcmp(XR HT->data[i].ex, id)) x_index = i;
    else if (!strcmp(XR HT->data[i].ex, ALL)) all_index = i;
    else if (!strcmp(XR HT->data[i].ex, FINAL)) final_index = i;
    
    f = Stack_ChrFrame ();
    strcpy (_EXN_LAST, id);

---

Exceptional C: design and implementation of the XTC compiler
EXN_ARG_LAST = args;
FROMRAISE = 1;

for (; ;) {
    Stack_PreviousFrame (&f, &f);
    if (f.pc == 0) {
        printf ("raise: no handler found for \%s\n", id);
        abort (0);
    }
    if ((x_index >= 0) &&
        Handler=I_E_Handler((ExmRegistry_CodePtr) f.pc, (ExmRegistry_Ranges *)
                        &XR_HT->data[x_index].ranges)) {
        break;
    }
    else if ((all_index >= 0) &&
              Handler=I_E_Handler((ExmRegistry_CodePtr) f.pc, (ExmRegistry_Ranges *)
                        &XR_HT->data[all_index].ranges)) {
        break;
    }
    else if ((final_index >= 0) &&
              Handler=I_E_Handler((ExmRegistry_CodePtr) f.pc, (ExmRegistry_Ranges *)
                        &XR_HT->data[final_index].ranges)) {
        break;
    }
}

Stack_Unwind (&f, (t_addr) handler);
printf ("raise: unwind returned!'\n");
abort (0);
}

char tmp[80] = "Global String will be used as argument\n";
int Globl = 100;

typedef struct ExArgs { char s[80]; int i; } Args;

class void inner_blow_up (unsigned int i)
{
    int *ip = &Globl;
    Args arg1;
    strncpy (arg1.s, tmp);
    arg1.i = 10;
    { int *x_ip1 = (int *) malloc ( sizeof(arg1) + sizeof(char *) +
                          sizeof(i) + sizeof(ip));
      int _x_ip2 = _x_ip1;
      memcpy ((void *)(_x_ip2 = _x_ip2+0), (void *)&arg1, sizeof(arg1));
      char _x_ip3 = "this is a const string arg";
      memcpy ((void *)(_x_ip2=_x_ip2+sizeof(arg1)), (void
*))(void *)(_x_ip2+sizeof(char*));
    }
    memcpy ((void *)(_x_ip2 = _x_ip2+sizeof(char*))), (void *)&arg1, sizeof(i));
    memcpy ((void *)(_x_ip2 = _x_ip2+sizeof(i))), (void *)&ip, sizeof(ip));
    _raise ("exception1", _x_ip1); }

printf ( " --- oops! after RAISE in inner_blow_up\n");
}

class void blow_up (unsigned int i)
{
TRY(0)
    printf ( "\tbefore inner_blow_up (i=%d)\n", i);
    inner_blow_up (i);
    printf ( "\tOops! after inner_blow_up\n");
    FINALLY (1, _dum);

EXCEPTIONAL C: DESIGN AND IMPLEMENTATION OF THE XTC COMPILER
```c

printf ("\tinner finally %d\n", i);
TRY(2)
    printf ("\t\t\t\t\t\t\t\t\tttry in finally\n");
    { int _x_ip1 = (int *) malloc ( sizeof(char *)); int _x_ip2 = _x_ip1;
        char _x_ip_str = "test arg";memcpy ((void *) (_x_ip2=(_x_ip2+0)),(void *)&(_x_ip_str,sizeof(char*)));} _raise ("e1", _x_ip1);

    CATCH (3, _dumm) char *error;
    memcpy((void *)&error,(void *)(char *)_dumm, sizeof(char*));

    printf ("\t\t\tcought args: %s\n", error);
_ENDTRY
_ENDTRYFINALLY
}
main (int argc, char *argv[])
{
    printf ("test starts\n");
    printf ( "try_test: Main: argv0=%d argv[0]=%s\n", argc, argv[0]);
TRY(6)
    printf ("try stmt\n");
    blow_up(10);

    CATCH (7, _dumm) Args arg1;
    char *arg2;
    int arg3;
    int *last;
    memcpy((void *)&arg1,(void *)((Arg2 *)_dumm), sizeof(Arg2));
    memcpy((void *)&arg2,(void *)((char *)(_dumm+_dumm+sizeof(Arg2))), sizeof(char*));
    memcpy((void *)&arg3,(void *)((int *)(_dumm+_dumm+sizeof(char*))), sizeof(int));
    memcpy((void *)&last,(void *)((int*)(_dumm+_dumm+sizeof(int))),sizeof(int*));

    { int i;
        printf ("\n\t\t\tsought exception 1 with args:\n");
        printf ("\t\targ1.s = %s\nstring was: %s\narg3 = %d\n\n last points to %d\n", arg1.s, arg1.i, arg2, arg3, *last);
    }
_ENDTRY

}```