Exception handling is a programming language feature that can help increase the reliability of programs. However, not much work has been done on exception handling in visual programming languages. We present an approach for improving the exception handling mechanism in Forms/3, a declarative visual programming language based on the spreadsheet paradigm. We show how this approach can be added without sacrificing referential transparency and lazy evaluation in Forms/3. We then present a comparison of the Forms/3 exception handling mechanism with the mechanisms available in Java, C++, Prograph, Haskell and Microsoft Excel, based on their expressive powers.
The Expressive Power and Declarative Attributes of Exception Handling in Forms/3

by

Anurag Agrawal

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed July 14, 1997
Commencement June 1998
Master of Science thesis of Anurag Agrawal presented on July 14, 1997

APPROVED:

Redacted for Privacy

Major Professor representing Computer Science

Redacted for Privacy

Head of Department of Computer Science

Redacted for Privacy

Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for Privacy

Anurag Agrawal, Author
ACKNOWLEDGMENT

I would like to thank my advisor Dr. Margaret Burnett without whose support and guidance this work would not have been possible. I would also like to thank all the members of the Visual Programming Languages Research Group at OSU for their contributions to the development of Forms/3 language and in helping to keep it up and running.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Basic concepts</td>
<td>2</td>
</tr>
<tr>
<td>Organization of this document</td>
<td>3</td>
</tr>
<tr>
<td>Related Work</td>
<td>4</td>
</tr>
<tr>
<td>Historical perspective</td>
<td>4</td>
</tr>
<tr>
<td>The error value model</td>
<td>4</td>
</tr>
<tr>
<td>The replacement value model</td>
<td>5</td>
</tr>
<tr>
<td>Exception handling in imperative languages</td>
<td>6</td>
</tr>
<tr>
<td>Exception handling in object-oriented languages</td>
<td>8</td>
</tr>
<tr>
<td>Exception handling in lazy functional languages</td>
<td>9</td>
</tr>
<tr>
<td>Exception handling in VPLs</td>
<td>11</td>
</tr>
<tr>
<td>The Forms/3 VPL</td>
<td>13</td>
</tr>
<tr>
<td>Exception handling in Forms/3</td>
<td>13</td>
</tr>
<tr>
<td>Error value exception handling</td>
<td>14</td>
</tr>
<tr>
<td>If-then-else + declarative semantics + output = rules</td>
<td>16</td>
</tr>
<tr>
<td>Creating user-defined exceptions using composition and abstraction</td>
<td>17</td>
</tr>
<tr>
<td>Problems of using the error value model in a lazy declarative language</td>
<td>20</td>
</tr>
<tr>
<td>Problem 1: Propagation of exceptions</td>
<td>20</td>
</tr>
<tr>
<td>Problem 2: Referential transparency in the presence of multiple exceptions</td>
<td>21</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Problem 3: Maintaining laziness</td>
<td>21</td>
</tr>
<tr>
<td>Forms/3’s solutions for error value model problems</td>
<td>23</td>
</tr>
<tr>
<td>Solution to problem 1 (propagation of errors):</td>
<td>23</td>
</tr>
<tr>
<td>Solution to problem 2 (referential transparency in the presence of multiple exceptions):</td>
<td>25</td>
</tr>
<tr>
<td>Solution to problem 3 (referential transparency in the presence of exceptions in lazy functions):</td>
<td>26</td>
</tr>
<tr>
<td>Comparison of Forms/3 exception handling mechanism</td>
<td>29</td>
</tr>
<tr>
<td>Expressive power of exception handling mechanisms</td>
<td>30</td>
</tr>
<tr>
<td>Forms/3</td>
<td>34</td>
</tr>
<tr>
<td>Microsoft Excel</td>
<td>43</td>
</tr>
<tr>
<td>Java</td>
<td>46</td>
</tr>
<tr>
<td>C++</td>
<td>53</td>
</tr>
<tr>
<td>Prograph</td>
<td>54</td>
</tr>
<tr>
<td>Haskell</td>
<td>62</td>
</tr>
<tr>
<td>Overall comparison</td>
<td>66</td>
</tr>
<tr>
<td>Conclusions</td>
<td>67</td>
</tr>
<tr>
<td>References</td>
<td>68</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>1.</td>
<td>Exception handling in PL/1</td>
</tr>
<tr>
<td>2.</td>
<td>User view of simple clock program with no exceptions</td>
</tr>
<tr>
<td>3.</td>
<td>User view of clock program with invalid input</td>
</tr>
<tr>
<td>4.</td>
<td>Programmer view of clock program</td>
</tr>
<tr>
<td>5.</td>
<td>Clock program with exception handling</td>
</tr>
<tr>
<td>6.</td>
<td>primitiveError form</td>
</tr>
<tr>
<td>7.</td>
<td>Multiple exceptions in Forms/3</td>
</tr>
<tr>
<td>8.</td>
<td>Multiple exceptions as a set in Forms/3</td>
</tr>
<tr>
<td>9.</td>
<td>Built-in exceptions in Forms/3</td>
</tr>
<tr>
<td>10.</td>
<td>User-defined exceptions in Forms/3</td>
</tr>
<tr>
<td>11.</td>
<td>Possible continuations in Forms/3</td>
</tr>
<tr>
<td>12.</td>
<td>Transferring information from signaler to handler in Forms/3</td>
</tr>
<tr>
<td>13.</td>
<td>Default exception handlers in Forms/3</td>
</tr>
<tr>
<td>14.</td>
<td>eventReceptor form in Forms/3</td>
</tr>
<tr>
<td>15.</td>
<td>Handling user input as exceptions in Forms/3</td>
</tr>
<tr>
<td>16.</td>
<td>Programming with the spreadsheet model in Microsoft Excel</td>
</tr>
<tr>
<td>17.</td>
<td>Exceptions in Microsoft Excel</td>
</tr>
<tr>
<td>18.</td>
<td>Exception handling in Java</td>
</tr>
<tr>
<td>19.</td>
<td>Grouping of exceptions in Java</td>
</tr>
</tbody>
</table>
**LIST OF FIGURES (Continued)**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.</td>
<td>Programming with the dataflow model in Prograph</td>
<td>55</td>
</tr>
<tr>
<td>21.</td>
<td>Controlling flow of execution using Next Case on Failure control in Prograph</td>
<td>56</td>
</tr>
<tr>
<td>22.</td>
<td>Controlling flow of execution using Next Case on Success control in Prograph</td>
<td>57</td>
</tr>
<tr>
<td>23.</td>
<td>Using Fail control in Prograph</td>
<td>58</td>
</tr>
<tr>
<td>24.</td>
<td>Signaling exceptions using Match operator in Prograph</td>
<td>59</td>
</tr>
<tr>
<td>25.</td>
<td>Signaling exceptions using boolean primitives in Prograph</td>
<td>60</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. or decision table</td>
<td>27</td>
</tr>
<tr>
<td>2. and decision table</td>
<td>28</td>
</tr>
<tr>
<td>3. if decision table</td>
<td>28</td>
</tr>
<tr>
<td>4. Expressive power of different exception handling mechanisms</td>
<td>66</td>
</tr>
</tbody>
</table>
DEDICATION

This thesis is dedicated to my mom and dad.
THE EXPRESSIVE POWER AND DECLARATIVE ATTRIBUTES OF
EXCEPTION HANDLING IN FORMS/3

Introduction

Until recently, exception handling in programming languages was a feature that did not attract much attention in the mainstream programming community. Most programming languages provided little or no support for it. Even in the languages that did provide an explicit exception handling mechanism, the constructs were obscure, and the compilers rarely checked to make sure that they were used. As a result, there was little motivation for programmers to use it.

All this has changed with the advent of Java. With its extensive support for exception handling, and strict compile time checking to ensure that exceptions are handled, programmers have no choice but to use exception handling in their code. This has focused much attention on exception handling as a software engineering tool.

We believe that exception handling is an important feature in making programs more robust and error free. We also believe that a well-implemented exception handling mechanism can make programs easier to understand and write by cleanly separating the algorithm part of a function from the code required to deal with exceptional circumstances. Because of this, we believe that exception handling is an important area for research, especially in visual programming languages (VPLs) which are designed to be easier to understand and program in. Surprisingly, though, exception handling has so far largely been ignored in most visual programming languages.

In this document, we present some enhancements to the exception handling mechanism in Forms/3 [Burnett and Ambler 1994], a visual programming language based on the spreadsheet paradigm, in a continuation of the research presented in [van Zee et al. 1996]. The evaluation model of Forms/3 is lazy and declarative. There are some important problems involved in incorporating exception handling in lazy declarative
languages in general. We describe these problems, and show how we resolved them in Forms/3. Finally, we present a comparison of the exception handling mechanism of Forms/3 with that used in representative languages from object oriented, functional and visual programming paradigms.

Finding a basis for comparing the exception handling mechanisms of different languages was not obvious because exception handling is generally closely tied to the evaluation model of a language, and it was difficult to consider only the exception handling mechanisms in isolation. In addition, some of the languages used different interpretations of what an exception handling mechanism was expected to do. To resolve this situation, we derived from the literature a set of criteria that compares only the expressive power of the different exception handling mechanisms. [Sebesta 1996] defines a language feature to have more expressive power if it provides "relatively convenient, rather than cumbersome, ways of specifying computations."

**Basic concepts**

Sebesta [Sebesta 1996] defines an exception to be any unusual event, erroneous or not, that is detectable by hardware or software and that may require special processing. Examples of exceptions are errors detected by hardware, such as disk read errors, or unusual conditions, such as end-of-file. Things like hardware interrupts and asynchronous user input can also be termed exceptions. An exception is signaled when its associated event occurs.

The special processing that may be required by the detection of an exception is called exception handling. The code unit that does this processing is called the exception handler.

Usually, different exception handlers are provided for different exception types. For example, the handling required to deal with an end-of-file exception is rather different from the handling required to deal with an arithmetic overflow exception.
However, there are cases where the only action done by the handler might be to generate an error message, and terminate the program in an orderly manner. In this case, the same exception handler can be used for different exception types.

Once the exception has been signaled, there are a number of possible ways that the exception handler can respond. [Yemeni and Berry 1985] identify five possible responses:

- Resume: do something, and then resume execution from the point where the exception was signaled.
- Terminate the signaler: do something, and then resume execution at the construct following the invocation of the operation.
- Retry the signaler: do something, and then invoke the signaler again, possibly with different parameters.
- Propagate the exception: Pass the exception on to the invoker of the invoker of the signaling operation. This might include signaling a different exception.
- Transfer control: Do something, then transfer control to another location in the program.

**Organization of this document**

We begin by discussing the historical background and related work. This is followed by an introduction to exception handling mechanisms of Forms/3, which is built upon what is known as error value exception handling. We then describe problems associated with incorporating error value exception handling in lazy declarative languages in general, followed by Forms/3 solutions to these problems. Finally, we provide an extensive comparison of the exception handling mechanism in Forms/3 with the exception handling mechanisms of Java/C++, Microsoft Excel, Prograph and Haskell, based on the expressive power of these mechanisms.
Related Work

**Historical perspective**

The first extensive piece of research on exception handling was presented in [Goodenough 1975]. The paper identified some useful issues relating to exception handling:

1) An exception's full significance is known only outside the detecting operation.
2) Exceptions are used primarily for two purposes:
   - to permit dealing with an operation's impending or actual failure
   - to indicate the significance of a valid result or the circumstances under which it was obtained, e.g. end of file or empty queue.

The language CLU [Liskov and Snyder 1979] included exception handling based on the ideas described by Goodenough. Most subsequent languages have based their exception handling mechanism on the CLU design. The CLU mechanism was based on the principle that exceptions are handled by the calling method of the procedure that signaled them. A method could signal an exception using an explicit `signal` command. This would terminate the method, and pass control to a handler specified at the end of the calling function's definition. If no handler for that exception was specified in the calling function, the program would terminate. After the exception was handled, the program would continue from the statement following the exception handler.

**The error value model**

A commonly used exception handling mechanism in spreadsheets is the error value model. Under this model, an operation returns a special value whenever it detects an exception. If no exceptional event occurs during the evaluation of the operation, the
operation returns the expected value after computation. For example, if the two arguments to a + operator are 3 and "a", the value returned by the application of + on these arguments could be TYPE-ERROR. i.e.

\[ 3 + "a" = \text{TYPE-ERROR} \]

However, if the arguments passed to the operator are valid, like 3 and 5, then the result is the expected sum of the two values. i.e.

\[ 3 + 5 = 8 \]

In a later chapter, we will show the use of the error value model in Forms/3 [Burnett and Ambler 1994], a visual programming language based on the spreadsheet model, and Microsoft Excel, a commercial spreadsheet.

**The replacement value model**

Yemini and Berry [Yemini and Berry 1985] presented an expression-oriented exception handling mechanism, called the replacement value model. In this mechanism, an exception handler is explicitly associated with the expression that can signal the exception. The handler's result replaces the result of the subexpression signaling the exception, or replaces the expression's result. This model can support all possible handler responses of continuation, namely resumption, termination, retry, exception propagation and transfer of control. While designing this mechanism, Yemini and Berry also came up with a set of design guidelines:

- Handlers should be allowed to have formal parameters.
- Exception handling should integrate fully with a language's scope rules and type system.

---

1Yemini and Berry name it the "replacement model". We use the phrase "replacement value model" in this paper because it emphasizes the nature of the approach and aligns nicely with the use of values in Forms/3.
• To preserve information hiding, unhandled exceptions should not automatically propagate along the chain of invokers.

• Exception handling features should be designed so that their addition to a language does not reduce the language's suitability for formal verification.

• Data and procedural abstractions should be able to include exceptions in their definitions.

The replacement value model was initially constructed for imperative languages. However, it has since been adapted to functional languages [Bretz and Ebert 1988] and visual programming languages [van Zee et al. 1996]. In [van Zee et al. 1996] the authors demonstrated how the replacement value model can be implemented in a declarative visual programming language by building upon the error value model.

**Exception handling in imperative languages**

The earliest languages to have exception handling were imperative languages, mainly because of the fact that most early languages were imperative. However, typical of the languages of that era, the exception handling mechanisms of those languages were often too complicated or too unstructured for programmers to use effectively. A prime example is PL/1.

PL/1 had 22 standard built-in exceptions and there were built-in exception handlers for each of these exceptions. In addition, users could define their own exceptions, and define handlers for both user-defined exceptions and built-in exceptions. User-defined exception handlers could be defined anywhere an executable statement could occur because they were themselves executable statements. A user-defined handler for an exception would stay in effect until it was overridden by a new handler (or the block in which it occurred exited).

The built-in exception handlers caused two different continuation actions. In some cases, they caused the program to terminate, and in some other cases they caused
control to return to the statement that signaled the exception. The user-defined exceptions could transfer control to anywhere in the program after handling the exception, but there was no mechanism that provided the address of the statement that signaled the exception. So, it was almost impossible for user-defined exceptions to return control to the statement that signaled the exception.

The exception handling mechanism of PL/1 was powerful and flexible. However, it was difficult to understand and use reliably because of the dynamic binding of handlers to exceptions and the flexibility of the continuation rules. For example, dynamic binding of the exceptions to handlers caused problems similar to those involved in dynamic scoping. Since the scope of the exception handler was dynamic, it was very difficult to determine from the source code which binding was in effect at a given point in the program. Consider the program in the Figure 1 [Sebesta 1996]:

```
(SUBSCRIPTRANGE):
   BEGIN;
   ...
   ON SUBSCRIPTRANGE
      BEGIN;
      PUT LIST ('ERROR - BAD SUBSCRIPT IN ARRAY SUBSUM');
      GO TO FIXIT;
      END;
   ...
   ON SUBSCRIPTRANGE
      BEGIN;
      PUT LIST ('ERROR - BAD SUBSCRIPT IN ARRAY BLK');
      GO TO QUIT;
      END;
   ...
LABEL1:;
   ...
   BLK (I, J, K) = SUM;
   ...
   END;
```

Figure 1: Exception handling in PL/1
(Here, the statement (SUBSCRIPTRANGE) indicates that the built-in exception SUBSCRIPTRANGE is enabled in this block. The group of statements

ON SUBSCRIPTRANGE

BEGIN

...

END

establish the handler for the exception SUBSCRIPTRANGE.)

If the code between the two handlers for the exception SUBSCRIPTRANGE happened to include a GO TO LABEL1, then the first handler would be executed if the exception was raised by the assignment to BLK. This would enact the wrong handler.

The software engineering lessons learned from the exception handling mechanism in PL/1 served as the basis on which [Goodenough 1975] developed the software engineering issues of exception handling, and these issues were in turn later the basis for the implementation of the exception handling mechanism in CLU.

**Exception handling in object-oriented languages**

Exception handling in most modern object oriented languages builds upon the exception handling mechanism research in imperative languages, but derives additional structure from the class hierarchy of object oriented languages. In this mechanism, the signaler of an exception "throws" an object, which might contain information relating to the cause of the exception. Exception handlers (established by a series of "catch" declarations) are defined inside the caller of the code that signals the exception. A handler is bound to an exception by the type or class of the object thrown by the signaler. The signaling of an exception causes an immediate termination of the code block which signaled the exception, and causes control to be transferred to the corresponding exception handler.
Some designers consider exception handling to be an integral part of a good object oriented design because signaling an exception is one of the ways an object interacts with the outside world. Since signaling exceptions is one of the ways a function might return a value to its caller, they believe that a function should include all the exceptions that it can signal as part of its external interface, just like the type of its return value and input parameters. This philosophy is evident in the design of Java where the language definition requires every function to include all the exceptions it can signal to be listed in its function declaration\(^2\). C++ also provides this facility, but does not enforce its use because of the need for compatibility with existing code written in C, which has no structured exception handling mechanism.

We include an analysis of Java and C++ exception handling mechanisms in the section comparing expressive power of exception handling mechanisms.

**Exception handling in lazy functional languages**

[Bretz and Ebert 1988] describe a method for adapting replacement value exception handling to functional languages. However, their mechanism does not allow lazy evaluation in the language. [Reeves et al. 1989] extends their mechanism to lazy functional languages, using firewalls to encapsulate the context of an exception handler for use during lazy evaluation. However, the mechanism described by Reeves et al. adds complexity to the language by requiring firewalls, and prioritization of exception handlers. The priorities of the exception handlers are established by the order in which

\(^2\)Except for a group of functions that are subtypes of class Error or RuntimeException. These exceptions are not required to be part of a function’s interface because these classes are reserved for exceptions that the designers felt are too common to be always explicitly handled, or are due to internal errors in the Java virtual machine, which the programmer cannot be expected to handle. However, even though they are not required to be part of the interface, these exceptions can be put in the function declaration if the programmer chooses to do so for the sake of completeness. In addition, the function is allowed to set up handlers for all of these exceptions.
the handlers were specified. This is unusual for a functional language, since generally, the order of execution of statements is not supposed to matter in most functional languages.

Haskell [Peterson et al. 1997a] is a lazy functional language that provides exception handling through the use of monads. However, the monadic approach to exception handling is limited and discourages programming in the pure functional style. (A description of Haskell exception handling is given in a later chapter.)

[Wadler 1985] describes a method that avoids introducing extra mechanisms to deal with exceptions in a lazy functional language by making use of lists. The paper describes a method under which a function that can either fail or return a value on success, instead returns a list of the (one) answer. Upon failure the list is empty, because there is no answer. This method solves the problem of exception handling by the fact that every function always results in a list. If the functions in the language are defined to take list input and return list outputs, then dealing with exceptions is simply reduced to dealing with an empty list. For example [Wadler 1985], consider a function assoc that looks up entries in an association list. That is, given a list of pairs xys, and a value x, the call assoc xys x returns a value y such that the pair (x, y) is in the list xys. If there is no such y, then the call should signal an exception. Thus,

```plaintext
assoc ["a", 1), ("b", 2]] "b" = 2
assoc ["a", 1), ("b", 2]] "c" = FAIL
```

where FAIL indicates that evaluation signaled an exception. This function could be written as:

```plaintext
assoc [] x = FAIL
assoc ((x', y)xys) x = y IF x' = x
assoc xys x OTHERWISE
```

Using lists, the function could instead be defined as:

```plaintext
assoc xys x = cut [y | (x', y) <- xys, x' = x]
```

where cut is defined by:

```plaintext
cut [ ] = []
```
cut (x:xs) = [x]

(Essentially, the function cut truncates a given list to have at most one element.)

Using this definition of assoc, we get,

assoc ["a", 1], ("b", 2) "b" = [2]
assoc ["a", 1], ("b", 2) "c" = []

The author combined this technique with backtracking to handle pattern matching in lazy functional languages. However this approach is not sufficiently exhaustive. The paper limits the term "exception" to mean only those situations where the function returns failure or NULL as the answer. Unusual or unexpected situations like division by zero, or end of file are not treated as exceptions. In addition, there is no way of determining the cause of the exception.

**Exception handling in VPLs**

Most visual programming languages do not include explicit exception handling mechanisms. One of the few papers to include a lengthy discussion of errors and exception handling in VPLs is [Cox et al. 1995]. In that paper, the authors describe exception handling in Prograph, a commercially available VPL. Exception handling in Prograph is tightly integrated with mechanisms for regulating control flow. We include an analysis of the Prograph exception handling mechanism in the section on expressive power.

The language Fabrik [Ingalls et al. 1988], a dataflow language, has limited exception handling to the extent that if a component is unable to compute, the values on the output pins become invalid, and this invalidity is passed on to the connected pins, overriding any default value defined for the pin. Connections that carry an invalid value are shown as a dashed line.

In [van Zee et al. 1996] the authors describe exception handling in Forms/3, a visual programming language based on the spreadsheet paradigm, and demonstrate how
replacement value exception handling could be incorporated to a declarative VPL using the error value model. The work presented in this document is a continuation of their research.
The Forms/3 VPL

Forms/3 is a general purpose, declarative VPL. Its goal is to provide computational and expressive power in a language featuring a simple, concrete programming style with immediate feedback. Programming in Forms/3 follows the spreadsheet paradigm; the programmer uses direct manipulation to place cells on forms, and then defines a formula for each cell. Forms are the basic organizational units, and cells are the computational units. Because each cell's value is determined by its formula, a program's behavior is entirely determined by the cells' formulas. Forms/3 is fully live, which means that it automatically re-evaluates on-screen values whenever a formula is changed or new data arrives.

Exception handling in Forms/3

Our primary objective was to incorporate a full-featured exception handling mechanism that fits seamlessly with the spreadsheet paradigm and the lazy evaluation model of Forms/3. This implied that we could not use things like throws and catches, or transfer of control, or imperative constructs that do not comply with the normal evaluation model of Forms/3.

To fulfill these goals, in Forms/3, we combine two approaches to exception handling, the error value model, and the replacement value model. The error value model constitutes the basic exception handling mechanism, on top of which the replacement value model is implemented, as was demonstrated in [van Zee et al. 1996]. The rest of this section gives a brief description of the error value exception handling in Forms/3.
Error value exception handling

The analog clock programs in Figure 2-4 demonstrate the error value model of exception handling in Forms/3. The clock program takes two integers, representing the time of day in hours and minutes, and displays the corresponding analog clock. The x- and y- positions of the clock's hands are computed by cells minutex, minuty, hourx, and houry. Cell theClock references the results of the cells minuteHand, hourHand, face and pivot to assemble the clock components into one unit. The combination of lazy evaluation with liveness in Forms/3 causes execution of formulas to be automatically scheduled for every cell that is currently on the screen, as well as for any other cells needed to compute those on-screen cells.

Because the programmer has not provided any exception handling code in the formulas in Figure 3, the program defaults to the exception handling automatically provided by the system under the error value model. Whenever the system detects an error in processing a formula it signals this exception by returning a value of type error. Each such error value has a displayable message string containing the cause of the exception. For example, in Figure 3, a character is entered as the formula for minute instead of an integer, and the system signals this by returning a "TYPE-ERROR".
Figure 2: User view of clock program with no exceptions.

Figure 3: User view of clock program with invalid input.
If-then-else + declarative semantics + output = rules

Programmers can handle error values by using ordinary if-then-else formulas. For example, suppose we rename theClock to goodClock, add a badClock cell containing a sketch of a broken clock (drawn using an ordinary X-Windows bitmap editor), and create a new theClock cell with formula:

if(error? (minuteHand) or error? (hourHand))
then badClock else goodClock

(The operator error? tests whether a value is of type error.) Figure 5 shows the result of these three changes.
As [van Zee et al. 1996] so aptly put it "This example illustrates the key characteristic of declarative languages that can be exploited to allow seamless exception handling under the error value model. The ordinary \texttt{if-then-else} conditional construct in a declarative language, when paired with a demand for output, provides the same functionality as rule-based semantics. This is because (1) the declarative nature of the language says that the variables' definition (in Forms/3, these are the cells' formulas) entirely define all the relationships in the program, and (2) wherever output is produced in such a language, the system must automatically maintain all values contributing to the output. This combination provides exactly the evaluation model needed for exception handling, because programmer-supplied exception handlers are the equivalent of rules that must be followed whenever the associated exceptions arise."

\textbf{Creating user-defined exceptions using composition and abstraction}

New exceptions can be created in Forms/3 using both the simple \texttt{if-then-else} construct and the built-in approach to data abstraction.
The **if-then-else** construct can be used to create new exceptions by capitalizing on the fact that all instances of the error type are just like any other value. For example, if the clock program referenced the system's clock rather than user input, we might add an 8:10 alarm using an **alarm** cell:

```plaintext
if (hourHand = 8) and (minuteHand = 10)
    then TRUE else FALSE
```

Other cells in the program could then refer to this cell in their own formulas (e.g., "**if alarm then ...**"). Such uses of if-then-else can involve arbitrarily complex combinations, and can result in values of any type, not just Booleans. This way of composing works with any kind of exceptions, whether errors or not.

In addition, Forms/3 also provides the ability to create exceptions of the type **error**. This facility might be useful in situations where the programmer wants to differentiate between exceptions that are errors and exceptions that are not.

Like every other data type in Forms/3, the type **error** is an abstract data type. To signal an exception with an instance of this type, the Forms/3 programmer may use the **error** operator (if minuteHand > 60 then error else...), which is a shortcut for a reference to cell **newError** on a copy of the primitive error (Figure 6). This form allows programmers to insert arbitrarily complex data into instances of the error type to define their own kinds of errors.
The replacement value model of exception handling in Forms/3 is built on top of the error value model, as was demonstrated in [van Zee et al. 1996]. In this document, we are focusing on the error value model, because it is the basis upon which all of the exception handling mechanism in Forms/3 rests.
Problems of using the error value model in a lazy declarative language

The error value model provides a simple method for reporting system generated errors. However, when using it in a lazy declarative language like Forms/3, several problems need to be addressed. One of the contributions of this work is the solution of these problems.

Problem 1: Propagation of exceptions

Consider an expression

\[(3 / 0) + 5\]

In evaluating this expression, the sub-expression in brackets is evaluated. Since the sub-expression involves a division by zero, its value will be `DIVISION-BY-ZERO`.

There are two ways that the calculation can continue after this is detected:

1) continue evaluation, and try to use `DIVIDE-BY-ZERO` to compute the result.

In this case, the final answer would be:

`DIVISION-BY-ZERO + 5 = TYPE-ERROR`

(since the system doesn’t know how to add an exception to a number).

2) return `DIVISION-BY-ZERO` as the final answer.

The problem with the first approach (not propagating the exceptions) is that the result is not very informative. The exception signaled can be different from the one that actually occurred, thus conveying misleading or insufficient information about the cause of the exception to the user of this expression.

The 2nd approach (propagating the exceptions) conveys the correct cause of the exception to the user, but adopting it causes new problems, which are discussed in the next section.
Problem 2: Referential transparency in the presence of multiple exceptions

Suppose that the second approach is adopted, and if a part of a formula generates an exception, then that value is reported as its result. Now, this raises the issue of referential transparency, which can be easily seen if there are multiple exceptions in the expression. For example suppose the expression is:

\[(3 / 0) + (5 + "a")\]

Two exceptions are generated during evaluation of this formula: DIVISION-BY-ZERO, and TYPE-ERROR. Which one should be reported as the result? It is possible to attach syntactic rules, based on the order of evaluation, like reporting the first exception generated while evaluating from left to right. However, this violates referential transparency, which states that mathematically identical expressions should always return identical answers. Referential transparency is one of the basic principles of a declarative language. For instance, if the left-most exception is reported first, the value of the above expression would be DIVIDE-BY-ZERO. However if the expression was changed to be

\[(5 + "a") + (3 / 0)\]

the result would be TYPE-ERROR. Even though the two expressions are mathematically identical, different evaluation orders would give different results.

Problem 3: Maintaining laziness

In a lazy language, only code segments that can affect the answer are evaluated. So, if a part of an expression doesn’t need to be evaluated to produce the answer, then it is not evaluated. Consider the expression:

True OR (x > y)

According to lazy evaluation, the moment the evaluation engine encounters True, it realizes that it has the result, and doesn’t need to evaluate the rest of the expression. Suppose that (x > y) generated an exception, say TYPE-ERROR. If the evaluation...
engine encounters True first, it will stop at that point, and the value of the expression will be set to True. However, consider the case that \((x > y)\) does get evaluated. If the exception handling mechanism requires reporting of all exceptions encountered, then the value of the expression would be **TYPE-ERROR**. (Note that there is no obvious way of guaranteeing that \((x > y)\) is never evaluated, because \(\text{True OR (x > y)}\) is mathematically identical to \((x > y) \text{ OR True}\). Due to the combination of laziness with the possibility of an exception, the evaluation model no longer would be referentially transparent.)
Forms/3's solutions for error value model problems

Solution to problem 1 (propagation of errors):

Exceptions of type error are propagated in Forms/3.

The occurrence of an exception implies that something different from what was expected occurred, and for the program to execute correctly, it has to handle this exception. The ideal case would be that the exception is handled at the point where it is generated. However, if it is not, either because the code at that level doesn't have enough information to handle it, or because the programmer forgot to handle it, ignoring the problem won't make it go away. For example, suppose cell A's formula is:

\[ A: (3 / 0) + 5 \]

Once \((3 / 0)\) is evaluated to \textbf{DIVIDE-BY-ZERO}, any computation using this value cannot produce a correct answer until the exception is handled. If the exception weren't propagated automatically, the expressions using this value would assume that the value was valid, and try to use it in further calculations. However, since the value is actually of type error, any mathematical operation on it would produce another exception, typically \textbf{TYPE-ERROR}. So, in a way, the exception would get distorted, and converted into something that no longer looks anything like the original exception. Thus, a cell separated a few levels from A would get an erroneous answer, but it wouldn't know what caused it, or possibly get misleading information about the cause. That is why it is better to propagate the actual exception.

[Yemini and Berry, 1985], state that "automatic propagation of unhandled exceptions may compromise information hiding, while explicit propagation can be used to properly rename propagated exceptions". In Forms/3, if the programmer of a library does want to maintain information hiding, then the exception can be handled within the
library. In case there isn’t enough information to handle the exception within the library, and the exception needs to be converted into another exception, that can be done explicitly by a handler in the library. For example, if a stack is implemented as an array, and we get an array_out_of_bounds exception. To hide the implementation from the user of the library, the library can have a handler for array_out_of_bounds exception, and return Stack Overflow to the user.

If it is left unhandled, the user gets an array_out_of_bounds exception. This is bad information hiding, but better than TYPE-ERROR, which leaves the user totally clueless.

We first tried propagating exceptions only up to a cell level, that is, exceptions generated during calculation of a cell’s formula were reported as the value of the cell. However, if the formula referenced some other cell whose value was of type error, the error was ignored, and the error value used in an attempt to calculate the answer. However, this approach didn’t work out well because it suffers from the same problems as earlier, and in addition it introduces additional problems with referential transparency. For example in the above example:

A: (3 / 0) + 20

The resulting value of the cell would be DIVIDE-BY-ZERO. Instead, suppose the formula was split up into two cells A and B with

B: (3 / 0), and
A: (B + 20).

If exceptions were not propagated, then B’s value would be DIVIDE-BY-ZERO, and A’s value would be:

"DIVIDE-BY-ZERO" + 20 = "TYPE-ERROR"

Instead, our approach returns the original error, and

"DIVIDE-BY-ZERO" + 20 = “DIVIDE-BY-ZERO”
Solution to problem 2 (referential transparency in the presence of multiple exceptions):

If evaluation of an expression generates multiple exceptions, its value is assigned to be a set composed of all of these exceptions. In Figure 7, the value of cell A is DIVISION-BY-ZERO, and that of cell B is TYPE-ERROR. The formula for cell C is:

\[ A + B \]

Since the values of both cell A and cell B are of type error, the value of cell C is set to be a union of the two errors.

The problem of referential transparency is solved simply because no matter in what order the parts as the expression are evaluated, all the exceptions generated are reported. The order in which the exceptions are discovered is not a concern because the returned value is a set, not a list. Figure 8 shows an example where the formula for cell D contains references to 3 cells A, B, C. The value of A is DIVISION-BY-ZERO, and the values of B and C are TYPE-ERROR. The resulting value of D is a set containing only one instance of both DIVISION-BY-ZERO and TYPE-ERROR.

![Figure 7: Multiple exceptions in Forms/3.](image-url)
Figure 8: Multiple exceptions as a set in Forms/3.

Solution to problem 3 (referential transparency in the presence of exceptions in lazy functions):

To maintain laziness we adopted the approach that if a part of an expression doesn't need to be evaluated to produce the answer, then it should not be evaluated. If it does get evaluated because of our implementation, the answer should be the same as if it had not been evaluated. So, for example, consider 2 cells with formulas:

A: TRUE or \((x > y)\)
B: \((x > y)\) or TRUE

The value of A and the value of B are both TRUE irrespective of whether \((x > y)\) generates an exception.

Why not return the exception in both cases? After all, this also returns the same answer in both cases. Because, then the system could no longer have lazy evaluation. When evaluating an expression, it would need to evaluate every subexpression solely to check for exceptions, even if they would otherwise not affect the answer.

There are three built-in operators in Forms/3 which are non-strict and can operate lazily - **if**, **or** and **and**. The following tables list the answers returned by these operators under different input conditions.
1) **or**: The operator **or** takes two arguments. It has the form 

(\text{argument1}) \textbf{or} (\text{argument2})

2) **operator and**: The operator **and** also takes two arguments, and has the form:

(\text{argument1}) \textbf{and} (\text{argument2})

3) **operator if**: The operator **if** takes three arguments and has the form:

if (\text{condition}) \textbf{then then_part else else_part}

Table 1 summarizes the answer returned by operator **or** under different conditions, and Tables 2 and 3 do the same for the operator **and** and **if** respectively. Rows correspond to different values for \textit{argument1}, and columns correspond to different values for \textit{argument2}. The value in each cell is the value returned for the corresponding values of \textit{argument1} and \textit{argument2}. The symmetry around the diagonal in Tables 1 and 2 are reflections of the referentially transparent semantics of these operations.

<table>
<thead>
<tr>
<th>argument 1 / argument 2</th>
<th>TRUE</th>
<th>FALSE</th>
<th>NON-BOOLEAN</th>
<th>EXCEPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>EXCEPTION</td>
<td>EXCEPTION</td>
</tr>
<tr>
<td>NON-BOOLEAN</td>
<td>TRUE</td>
<td>EXCEPTION</td>
<td>MULTIPLE-EXCEPTION</td>
<td>EXCEPTION</td>
</tr>
<tr>
<td>EXCEPTION</td>
<td>TRUE</td>
<td>EXCEPTION</td>
<td>EXCEPTION</td>
<td>MULTIPLE-EXCEPTION</td>
</tr>
</tbody>
</table>

Table 1: **or** decision table
<table>
<thead>
<tr>
<th>argument 1 / argument 2</th>
<th>TRUE</th>
<th>FALSE</th>
<th>NON-BOOLEAN</th>
<th>EXCEPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>EXCEPTION</td>
<td>EXCEPTION</td>
</tr>
<tr>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>NON-BOOLEAN</td>
<td>EXCEPTION</td>
<td>FALSE</td>
<td>MULTIPLE-EXCEPTION</td>
<td>EXCEPTION</td>
</tr>
<tr>
<td>EXCEPTION</td>
<td>EXCEPTION</td>
<td>FALSE</td>
<td>EXCEPTION</td>
<td>MULTIPLE-EXCEPTION</td>
</tr>
</tbody>
</table>

Table 2: and decision table.

<table>
<thead>
<tr>
<th>Value of condition</th>
<th>TRUE</th>
<th>FALSE</th>
<th>NON-BOOLEAN</th>
<th>EXCEPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value returned</td>
<td>then_part</td>
<td>else_part</td>
<td>EXCEPTION</td>
<td>EXCEPTION</td>
</tr>
</tbody>
</table>

Table 3: if decision table.
Comparison of Forms/3 exception handling mechanism

We believe that the exception handling mechanism in Forms/3 is quite comprehensive and useful, especially in the context of a declarative visual programming language with liveness as one of its principle characteristics. However, to verify its comprehensiveness, and to see if we had missed something, we believed that it was important to compare the Forms/3 exception handling mechanism with the exception handling mechanisms in other available languages.

Our first approach to performing the comparison was to try to prove the functional equivalence of the Forms/3 exception handling mechanism to the exception handling mechanisms in other languages. Proving functional equivalence implied that we needed to prove that we could completely emulate the exception handling mechanism of any of these languages using only the exception handling mechanism in Forms/3, and vice versa. Even if we could only prove that the exception handling mechanism of other languages could be emulated using the Forms/3 exception handling mechanism, it would prove that the exception handling mechanism of Forms/3 is at least as powerful as others. However, after some exploration we realized that there wasn’t a simple way of doing this. The exception handling mechanisms generally tended to be tightly integrated with the normal evaluation model of the languages they were embedded in, and it was difficult to consider only the constructs used for the exception handling mechanism in isolation.

This does not imply that the functionality achieved by using the exception handling mechanism in another language cannot be emulated in Forms/3. Indeed, since Forms/3 is Turing complete, it is possible to emulate in Forms/3 any functionality implemented in any other language. However, this does not prove the functional equivalence of the two exception handling mechanisms, as the functionality wasn’t achieved using only the exception handling mechanism of Forms/3.
Trying to prove equivalence in the opposite direction (implementing Forms/3 exception handling mechanism using another language's exception handling mechanism) also produced similar difficulties.

Instead, we adopted a different approach based on the expressive power of the exception handling mechanisms in different languages. This approach is described next.

**Expressive power of exception handling mechanisms**

According to [Sebesta 1996] a language construct has more expressive power if it provides "relatively convenient, rather than cumbersome, ways of specifying computations". Exception handling in a language is designed to improve the robustness and clarity of programs written in that language. Thus, to be effective, it has to be "convenient, rather than cumbersome". That is why we believe that comparing the expressive powers of the exception handling mechanisms is a reasonable way of comparing their effectiveness.

In order to do this comparison, we first had to derive a set of criteria that refer to the expressive power of an exception handling mechanism.

We derived the following criteria for comparing the expressive power of exception handling mechanisms from existing literature on exception handling and books on programming languages like [Sebesta 1996]. These criteria are derived largely by considering the exception handling features reported in the literature that are related to expressiveness. For each criterion listed, we discuss how it relates to expressive power.

1) **Is there support for any built-in exceptions?**: Built-in exceptions are exceptions that are part of the language definition, and can be detected and signaled by the runtime system for that language. Generally, if a language supports built-in exceptions, these exceptions include **hardware-detectable** exceptions (exceptions that can be detected by the hardware, like division by zero, or register overflow), and some software detectable exceptions like attempting to access an array element out the bounds
of the array. These exceptions can be important as they can signal potential causes of bugs or errors in the program. However, a number of languages that provide built-in exceptions (e.g., Java), do not provide support for hardware interrupts like alarms and I/O interrupts, which technically might also be possibly considered as hardware-detectable exceptions. However, these exceptions are covered by the question of whether the exception handling mechanism of a language supports asynchronous events. This issue is dealt with below.

2) Does the exception handling mechanism support specification of user-defined exceptions?: User-defined exceptions are new exceptions that the programmer defines in a program, and are not part of the language definition. These exceptions might give the programmer the ability to detect unusual conditions within their code, and signal exceptions corresponding to those conditions. This type of exception might also be used by a predefined library to signal exceptional circumstances to the user of the library.

3) Continuation - (how does execution continue after the exception handler for an exception finishes?): [Yemeni and Berry 1985] identify five ways a program can continue after an exception handler finishes - retry execution (re-invoke the module that signaled the exception), resume execution (restart execution at the point that the exception was generated), terminate execution (terminate the module that generated the exception), propagate the exception (pass the exception up to the caller), or transfer control to a new location.

We feel that terminate and resume responses could be used as a criterion for comparing the expressive power of a language, as some language designers consider these to be important parts of an exception handing mechanism, and most exception handling mechanisms give a reason for why one of these responses is available or not in that language.

The other three responses might not be good indicators of expressive power as they are either not universally applicable, or could be easily emulated. Specifically:
• transfer of control is not an issue in declarative languages, as there is no explicit flow of control. Also, in most imperative languages, it is possible to transfer control from within a piece of code to another location. So, it would almost always be possible to transfer control to some location in an imperative language, if a handler chooses to.

• propagation of the exception is so simply done manually in most exception handling constructs that it need not be automatic. In fact, some designers of exception handling, e.g. [Yemeni and Berry 1985], consider automatic propagation as a violation of information hiding, and disallow automatic propagation. On the other hand, other designers [Koenig and Stroustrup 1990] feel that automatic propagation removes the need for excess code, so automatic propagation is considered an important feature of languages like C++ and Java. However, this issue has little effect the expressive power of the exception handling mechanism very much, since if the system does not do automatic propagation, propagation can be done explicitly in the handler by simply signaling the exception again.

• Similarly, retry can also be easily emulated. If an exception causes the function throwing the exception to be terminated, the calling function can re-invoke that function by calling it again. If the signaling function is always resumed after the exception is handled, it can explicitly exit, and the calling function can re-invoke it.

4) Is it possible to convey information about the exception from the signaling point to the exception handler?: This issue could be important because this provides the exception handler with more information about the cause of the exception. The exception handler could then possibly be better equipped to deal with a specific exception.

5) Is it possible to have default handlers for exceptions?: Default-handlers are universal handlers that get invoked if the program does not establish explicit exception
handlers for an exception, and allow the program to continue after the exception has been handled. The ability to have default exception handlers can provide the possibility that an exception gets handled even if the programmer forgot to install a handler for it in the program.

6) Can the exception handling mechanism handle asynchronous events like mouse-events?: The ability to do so could provide a single mechanism for handling unusual conditions generated within the code, asynchronous user input and hardware events.

7) Is it possible to group exceptions and provide handlers for such a group?: This could make it possible for a single handler to cope with all exceptions coming from a major subsystem such as the file system, or the network subsystem, that might rely on hundreds of exceptions. The intention of grouping of exceptions is to allow exceptions to be divided into logical subdivisions, so that all exceptions within a subdivision can be handled together if required, but can also be handled specifically. There are two possibilities:

- grouping of multiple instances of one type.
- grouping of multiple types of exceptions.

In addition we believe that it might also be important to look at another question:

8) Is the Exception Handling mechanism enforced - that is, is the programmer required to provide handlers for all exceptions that can be thrown?

This question is not strictly an expressive power issue, but rather whether that power is utilized. However, it can make a big difference while coding in that language, since the underlying libraries may not provide any exception handling. On the other hand, if the exception handling mechanism is added on top of an existing language with large volumes of predefined libraries, it is impractical to require all code to comply with the mechanism, as that would require all existing code to be rewritten. So, we will treat it as a separate issue.
In following sections, we give a categorization according to these criteria of the most recent exception handling mechanisms using representative languages from object oriented, functional, and visual paradigms.

**Forms/3**

**Built-in exceptions:**

All low level exceptions (e.g.: division by zero, stack overflow, etc.) are built-in in Forms/3, and they are signaled by returning error values. For example, in the program shown in Figure 9, there are three cells, x, y, and z, whose formulas are

\[
\begin{align*}
x & : 3 \\
y & : 0 \\
z & : x / y
\end{align*}
\]

In the absence of any user-defined exception handling, the system returns a **DIVIDE-BY-ZERO** exception for the value of cell z.

![Figure 9: Built-in exceptions in Forms/3.](image)
User-defined exceptions:

Any combination of values can be combined to identify an exception in Forms/3. For example, the program in Figure 10 accepts a person’s age as user input. If the age entered is out of range (say, less than zero or greater than 125), or of the wrong type (if the user entered a string instead of a number, for example), an exception is generated, and the cell Message prompts the user with an error message. In this example, the user-defined exceptions are Constraint_Error and Data_Error. Cell Message defines the exception handlers for these two exceptions.

![User-defined exceptions in Forms/3.](image)

Figure 10: User-defined exceptions in Forms/3.

The additional cell exception is needed because Forms/3 does not have a built-in operator for checking types (this is because although Forms/3 is defined to have statically derived type checking, that is not implemented yet and the current system has to rely on
dynamic type checking). In this example, if the value of cell Age is non-numeric, performing the mathematical operation on it in cell exception signals a TYPE-ERROR, which can be detected using the error? operator.

Continuation:

It is possible to have both resume and terminate continuations in Forms/3, and the caller of a function can specify which form of continuation is actually taken when an exception is generated. Figure 11 shows a recursive factorial program, with replacement value exception handling. In replacement value exception handling, handlers are defined inside the callee, and the caller sets up the parameters that dictate the handlers' behavior.

The handler parameters are cells replacementValue and Mode. Cell replacementValue specifies the expression to be used as a substitute value if an exception is signaled. Cell mode is a radio button group used to specify the exception handling mode. (A radio button group in Forms/3 is a robust shortcut for a cell whose formula is intended to be one of an enumerated set of constants). A programmer sets the mode parameter simply by pushing one of these buttons. The three explicitly supported modes are:

- **Terminate**: If an exception is signaled, use the replacement value as a substitute for the final output cell(s) on the form.
- **Resume**: If an exception is signaled, use the replacement value as a substitute for the cell that caused the exception.
- **Retry**: If an exception is signaled, use the replacement value as a substitute for the initial input cell(s) on another invocation of the form.

(The other 2 possible continuations - propagation of the exception and transfer of control - are superseded. Propagation can be achieved by the handler itself signaling the exception. Transfer of control is not applicable to declarative approaches.)
Passing information from the signaling point to the handler:

Information about every exception except those that are in hidden cells (like "private" variables in C++ are available) to the handler in Forms/3. The program in Figure 12 classifies a stream of grades that are input through cell `newGrade` into ten ranges. The cells 0-9, 10-19 etc. show the count of the number of students with grades in that range. The program signals an exception (by setting cell `exception` to True) if a grade is greater than or equal to zero. Cell 90-100 handles the exception by checking to see if the input grade was exactly 100, and if so, increment its count. Otherwise, the grade is ignored. Here, information regarding the exception is available to the handler.
It is also possible to package up information about an exception via composition using the built-in error type (as shown in the previous section) or a new user-defined type. This returns information about the exception even when it is propagated away from its original source.

Figure 12: Transferring information from signaler to handler in Forms/3.

**Default handlers:**

The default handling for exceptions in Forms/3 is to display the exception. Barring bugs in the implementation of Forms/3 itself, it never crashes out of the application. For example, in the program in Figure 13, cell **Sum**'s formula is:

\[ x + y \]
However, the value of cell y is "a". Since cell Sum doesn’t know how to add a number to a string, a **TYPE-ERROR** exception is generated. In the absence of any programmer defined exception handling, the displayed value of cell Sum is set to the exception message, **TYPE-ERROR**.

![Figure 13: Default exception handlers in Forms/3.](image)

**Asynchronous events:**

Asynchronous events are handled with the exception handling mechanism in Forms/3. This is due to the fact that asynchronous events, just like all other exceptions, are signaled as values that can be simply “watched for” via the “rule-based” evaluation model that makes the if expression so powerful in this paradigm. This approach is unusual, since most languages (and all the other ones considered in this document) require separate handling mechanisms for exceptions and asynchronous events.

For example, to access mouse events, the user can use the `eventReceptor` form (Figure 14). This form allows the user to specify the events of interest, and provides information about the event when it occurs. The user specifies the events to be tracked using the `eventsOfInterest` cell ("leftdown" in this case, which stands for left mouse
button down event). The user specifies shape of the event sensitive area using cell \texttt{shape}. The cell \texttt{eventReceptor} creates an invisible area of this shape that will track the specified events. Whenever an event is detected, the cells on the form provide information regarding the event. For example, \texttt{x-position} and \texttt{y-position} provide the position of the cursor, \texttt{whatEvent?} indicates the name of the event that just occurred.

Application programs can handle events by referencing the cells on this form just like cells on any other form. In the example shown in Figure 15, the little circle in the cell \texttt{EventSensitiveArea} goes wherever the left mouse button is clicked. The shape of the event sensitive area is defined using cell \texttt{EventSensitiveShape}, and setting the formula of cell \texttt{shape} on the eventReceptor form to be a reference to this cell. Finally, the playing area is created in the cell \texttt{EventSensitiveArea}. When a user clicks the left mouse button inside the \texttt{EventSensitiveArea}, the event is detected, and the values of \texttt{x-position} and \texttt{y-position} change to the coordinates of the mouse cursor. This causes the little circle to move.
Figure 14: eventReceptor form in Forms/3.
Grouping of exceptions:

Grouping of exceptions is possible to some extent in Forms/3, as the language provides the ability to create abstract data types (called VADTs, which stands for Visual Abstract Data Type) [Burnett and Ambler 1994]. A new type of exceptions can be created by defining a new VADT, and then instantiating exceptions from this base type. All the exceptions of such a type can then be handled by simply specifying a handler for the VADT. Using these mechanisms, Forms/3 can group only exceptions that are instances of the same type. Grouping of exceptions that are of different types is not available in Forms/3.
Enforcement:

The exception handling mechanism is not enforced in Forms/3. The system does not check whether exceptions that can be signaled within a program are explicitly handled.

Microsoft Excel

Microsoft Excel is representative of conventional commercial spreadsheets. Since Forms/3 is also based on the spreadsheet paradigm, we believe that it is useful to compare its exception handling mechanism with that of Microsoft Excel.

Like most spreadsheets, an Excel worksheet consists of cells whose values can be entered directly, or be the result of formulas. The formula for a cell can be a reference to another cell, or contain operators and references to other cells. For example, in Figure 16 the value 10 is entered directly in cell A1. Cell A2’s formula is \texttt{IF(A1 > 0, True, False)}.

![Figure 16: Programming with the spreadsheet model in Microsoft Excel.](image)

Exception handling in Excel is based on the error value model and is similar to the error value exception handling in Forms/3. The system displays an error value in a cell when the formula for that cell cannot be calculated properly. If a formula includes a reference to a cell that contains an error value, that formula also produces an error value.
(except for the worksheet functions `ISERR`, `ISERROR`, or `ISNA`, which are used for detecting error values).

There are 7 predefined error values. These values are returned whenever the system detects an error. These exceptions can also be explicitly signaled by either entering them directly in a cell, or returning them as the result of a formula. For example, in Figure 17, the formula for cell A1 is set to 5 / 0. This causes the system to return #DIV/0! as its value. The formula for cell A3 is `IF(A2 < 100, A2, #NUM!)`. (the syntax of the IF formula is `IF(condition, value_if_true, value_if_false)` ). Here, if the value of A2 is greater than 100, A3 is set to the exception #NUM!.

![Figure 17: Exceptions in Microsoft Excel.](image)

A major difference between the Forms/3 and Microsoft Excel languages is the support for abstraction [Yang and Burnett 1994] and recursion in Forms/3. In addition there is a difference in granularity between the two language - in Forms/3 both a single cell and an entire form can be considered to be units of code, but in Microsoft Excel, only a single cell is the unit of code.

The major difference between the two exception handling mechanisms is that Forms/3 provides a customizable error type (using the primitiveError form), which Microsoft Excel does not. Combined with the data abstraction facility of Forms/3, this provides the ability to create user defined exceptions, and the ability to group exceptions.
(under the error type) in the Forms/3 exception handling mechanism. This ability is not available in the exception handling mechanism of Microsoft Excel.

We have listed below the expressive power criteria only for the aspects that are different for Microsoft Excel from Forms/3.

**Transferring information from signaling point to handler:**

It is not possible to transfer information through composition from the point where exception occurred to the exception handler, since the system allows only the exception type to be returned, with no additional information. Thus, the only mechanism for information transfer is to reference the original source of the exception.

**Asynchronous events:**

Asynchronous events cannot be handled using the exception handling mechanism in Microsoft Excel. The system provides a separate mechanism for including Visual Basic code to handle asynchronous events like mouse events.

**Grouping of exceptions:**

Grouping of exceptions of either type is not available in Microsoft Excel. The system only recognizes the seven distinct predefined exceptions. These exceptions are not arranged in a type hierarchy, and there is no mechanism to group them on the basis of types, unlike Java or C++ (which is described in the next section). In addition, there is no provision to create abstract data types in Microsoft Excel, unlike Forms/3. So, it is not possible to create exceptions that are different instances of a type, and then group them by defining a handler for the type.
Java

The exception handling mechanism of Java is representative of the approach adopted in recent object oriented languages. Java has focused a considerable amount of interest in the area of exception handling, as it is the first widely used programming language that requires exception handling constructs to be used in user programs.

Exceptions can be signaled in Java by using the explicit statement `throw`, whose general form is:

```java
throw [expression]
```

(The brackets are metasymbols used to specify that the expression is optional.) A throw without an operand can only appear in a handler. When it appears there, it resignals the exception, which is then handled elsewhere.

The type of the throw expression selects the particular handler, which must have a "matching" type formal parameter. In this case, matching means the following: A handler with a formal parameter of type T, matches a throw with an expression of type T, or any subclass of T. All exceptions in Java are subclassed from class Throwable.

In addition, exceptions can also be signaled by the Java Virtual Machine whenever it encounters a runtime error while executing the program. This includes things like division by zero, I/O errors, etc.

Exceptions handlers are specified in Java using the compound `catch` statement which appears after a compound `try` statement that contains the code that can signal exceptions. The general form of these two compound statements is

```java
try {
    --code that is expected to signal an exception--
}
catch (formal parameter) {
    --handler body--
}
```

...  
catch (formal parameter) {
    --handler body--
}
There can be only a single formal parameter, which is similar to a formal parameter in a function definition in Java. The handlers, as specified by the catch clause can include any Java code. All handlers have the same name (catch), but each must have a unique parameter type. A handler with a formal parameter of type Exception is the catch-all handler; it is enacted for any raised exception, if no more specific handler is chosen.

An exception raised in a try construct causes an immediate end to the execution of the code in that try construct. The search for a matching handler begins with the handlers that immediately follow the try construct. The matching process is done sequentially on the handlers until a match is found. In case no match is found locally, the exception is propagated to the caller of the function in which the exception is raised, and then to its caller, and so on. If no matching handler is found in the program, the program is terminated.

The finally clause is optional. However, if a finally clause is attached to a try...catch construct, the code in the finally block is always executed, regardless of whether the code in the try block signals an exception or not. If the code in the try block signals an exception, and the exception is caught by one of the catch clauses attached to the try block, then the code in the finally block is executed after the catch clause finishes. However, if the exception is not caught, and is propagated, or the catch clause itself throws an exception, then the code in the finally block is executed during the propagation of the exception, even if no matching catch clause is ultimately found. finally clauses are especially useful for resource deallocation, such as closing windows, releasing files, etc.
**Built-in exceptions:**

Java provides built-in exceptions for the following conditions:

- evaluation of an expression violates the normal semantics of the Java language, such as an integer divide by zero.
- an error occurs in loading or linking part of the Java program.
- some limitation on a resource is exceeded, such as using too much memory.
- the method `stop` of class `Thread` was invoked.
- an internal error occurs in the virtual machine.

(In fact, the Java language is designed to signal exceptions for almost every kind of error that can occur while executing a program, and a program is never supposed to crash without signaling an exception.)

**User-defined exceptions:**

It is possible to have user defined exceptions in Java. A new user-defined exception type is created by defining a new class derived from built-in class `Exception`, or any of its subclasses. For example in the program in Figure 18 [Gosling et al. 1996], a new exception type is created as class `TestException`. The `main` method of class `Test` invokes method `thrower` for each command line argument passed to the function. The function `thrower` causes different types of exceptions to be signaled, depending upon the command line arguments, all of which are caught by the `catch` block in function `main`. The catch block handles the exception by printing out a corresponding message. The code in the `finally` block in function `thrower`, is executed, regardless of whether an exception is signaled or not by the `try` block. For instance, if the command line argument to the program is `test`, the function `thrower` signals a `TestException`, and the output of the program is:

```
[thrower ("test") done]
```
Test "test" threw a class TestException
with message: Test message

class TestException extends Exception {
    TestException () { super (); }
    TestException (string s) { super (s); }
}

class Test {
    public static void main (String [] args) {
        for (int i = 0; i < args.length; i++) {
            try {
                thrower (args[i]);
                System.out.println("Test " + args[i] + " didn’t throw an exception");
            } catch (Exception e) {
                System.out.println("test " + args[i] + " threw a " + e.getClass() + " with message: " + e.getMessage())
            }
        }
    }
}

static int thrower (String s) throws TestException {
    try {
        if (s.equals("divide")) {
            int i = 0;
            return i/i;
        }
        if (s.equals("null")) {
            s = null;
            return s.length();
        }
        if (s.equals("test"))
            throw new TestException("Test message");
        return 0;
    } finally {
        System.out.println("[thrower " + s + "] done");
    }
}

Figure 18: Exception handling in Java.
Continuation:

Only terminate continuation (and not resume continuation) is available in Java. This is because when an exception is signaled in Java, the current block is immediately terminated, and control is always transferred to a handler which is defined outside of that block. There is no mechanism to restart execution at the point where the exception was signaled, except to re-invoke the entire block which is the same as retry continuation.

Transferring information from signaling point to handler:

It is possible to transfer information from the signaling point to the exception handler in Java. For example in Figure 18, when function thrower signals exception TestException, it passes the string "Test Message" to the exception object. The exception handler specified in the catch block in function main is able to access this message using the member function getMessage for this exception.

In general, since a new exception class is just like any other class in Java, it can have any number of data members, which can be set by the signaler of an exception, and accessed by the handler. So, the signaler has the ability to transfer an arbitrary amount of information to the handler.

Default handlers:

There are no default handlers for exceptions in Java. If an exception is signaled during the execution of a thread, and no handler is specified for it, that thread is terminated.
Asynchronous events:

Asynchronous events like mouse events are not handled in general by the exception handling mechanism of Java. The Java language provides a different mechanism under the Abstract Window Toolkit to handle user interface events.

The Java exception handling mechanism does support one asynchronous event - an explicit stop signal sent to a thread from another thread. Normally, a stop signal to a thread causes that the execution of that thread to be halted, and this exception is not generally handled explicitly.

Grouping of exceptions:

Grouping of exceptions that are of different types is implementable in Java with the use of inheritance. The exceptions that need to be in one group can all be subclassed from one base class. All the exceptions of a group can be handled with a single handler, by defining a catch block with the base class of that group as its formal parameter. In the example of Figure 19 [Gosling et al. 1996], exception classes TooHot and TooTired are derived from Grumpy. The try block can signal both TooHot and TooTired exceptions. Both of these exceptions are handled by the same catch block, whose formal parameter is of type Grumpy.

Grouping of exceptions that are different instances of the same type is available by defining a catch handler whose formal parameter is of that type.
class Grumpy extends Exception {}
class TooHot   extends Grumpy {}
class TooTired extends Grumpy {}

try {
    if ( temp > 40) throw (new TooHot ());
    if ( sleep < 8 ) throw (new TooTired ());
} catch (Grumpy g) {
    if (g instanceof TooHot)
        {System.out.println ("caught too hot!"); return;}
    if (g instanceof TooTired)
        {System.out.println ("caught too tired!"); return;}
}

Figure 19: Grouping of exceptions in Java.

Enforcement:

The Java compiler enforces the use of the exception handling mechanism by requiring that if the code within a function can signal an exception, then the function must either provide handlers for that exception, or it must include the exceptions in the function declaration. A function indicates that it can signal an exception by adding the keyword throws and the exception name after the name of the function in its declaration. The general form of such a function definition is:

    modifiers_and_returntype name (params) throws e1, e2, e3 {}

However, Java does make an exception to this rule for a group of exceptions which are subtypes of classes Error and RuntimeException. The compiler does not enforce the handling of these exceptions because these classes represent exceptions that the designers of the language felt were either too common, or indicated an internal Java machine error that the user program could not be expected to handle.
C++

The C++ exception handling mechanism [Koenig and Stroustrup 1990] is very similar to the Java exception handling mechanism. However the awareness about the two exception handling mechanisms is hugely different, with most Java programmers at least aware of the Java exception handling mechanism, and very few C++ programmers having any knowledge about the C++ mechanism. It is interesting to study the few differences between the two mechanisms, that may have caused this disparity.

The constructs of the C++ exception handling mechanisms are almost identical to the Java mechanism. Exceptions are signaled using throw statements, and handled using catch blocks attached to try blocks. Just like in Java, the throw statement returns an object to the handler, and the exception is handled by a catch statement whose formal parameter is of the same class or a superclass. However, C++ does not have a finally block.

The only differences in our criteria between the two mechanisms are listed below:

Built-in exceptions:

No built-in exceptions are defined in the C++ language.

Enforcement:

The C++ compiler does not check statically whether a program handles the exceptions that can be signaled during its execution. This is mainly because C++ is designed to be compatible with C, which does not support this exception handling mechanism. Forcing all code to handle exceptions would require a rewrite of all existing C and C++ code not having support for exception handling.
However, a programmer has the facility to handle exceptions within a function, or to declare the fact that it may signal an exception by using the `throws` keyword, like in Java.

**Other comments:**

A look at these two differences provides a hint to a possible cause for the disparity in the usage of the two exception handling mechanisms. The absence of built-in exceptions reduces the number of exceptional circumstances that can be handled by the programmer using the exception handling mechanism. (For example, events like file system errors are not signaled as exceptions in C++.) When this is combined with the lack of compiler enforcement, there might not be enough motivation for a programmer to use the exception handling mechanism in C++. (However, this situation might change when predefined libraries start shipping with built-in exceptions.)

**Prograph**

Prograph [Cox et al. 1995] is the most widely known commercially available general purpose VPL. Prograph is based on the data flow and object oriented paradigms. An important feature of it is the tight integration of the language with the editor, interpreter and debugger. The language also has a compiler that is meant to be used when the development phase of the program is finished.

The programmer codes by constructing drawings and the Prograph interpreter and compiler execute those drawings. Figure 20 shows a simple Prograph example that prompts the user for a number, and displays its square root. Each operation is represented by an icon, and data flow is represented by lines connecting the operations. An operation can execute whenever all the data input to it is available.
The exception handling mechanism of Prograph is built into its control flow mechanism. This works as follows:

Instead of an explicit if_then_else construct, Prograph has a case structure similar to case statements found in textual languages. The form of case constructs in traditional (textual) languages is:

```
CASE <selectorLabel> OF
  <label1>:<response1>;
  <label2>:<response2>;
  <label3>:<response3>;
END
```

The Prograph case structure is similar: if the condition is satisfied, a different dataflow diagram is executed, or an exception is raised. This construct is used both for conditional execution control flow (if-then-else in other languages) and for exception handling. The simplest form is the Next Case annotation with a conditional test based on boolean primitives. Next Case annotation has two forms: Next Case on Failure (Figure
21) and Next Case on Success (Figure 22). The control annotations are the small square boxes to the right of an operation, with a

![X]

in the box implying Next Case on Failure, and a

![✓]

implying Next Case on Success. A Next Case on Failure means "If this test fails, go to the next case". In the example in the left side of Figure 21, if the input is not greater than 21, the operation fails, and the next case (right side of the figure) is executed. Figure 22 is a similar program, but uses Next Case on Success instead of Next Case on Failure.

![Diagram](image)

Figure 21: Controlling flow of execution using Next Case on Failure control in Prograph.
To signal a failure, Prograph supplies the **Fail** annotation which signals the Failure exception. Upon failing, the called method notifies its calling operation that the called method has failed and terminates. How the calling operation reacts to the failure (handles the exception) is a function of the control annotation on the calling icon.

The **Fail** control can depend on failure or success:

- **Fail on Failure**,
  - 
- **Fail on Success**
  -

The example in Figure 23 shows the use of a **Fail** control. The function **search list for item** accepts a list input and a search value. The exit condition under which the function needs to stop iterating through the input list is when **search list for item** finds an item in the list that matches the search item. So, **search list for item** is to fail when it succeeds in finding the search item. That is why the test for equality between the item to
be searched and the search item in the first case of search list for item has a Fail on Success control.

Figure 23: Using Fail control in Prograph.

Since the list itself may be composed of lists, the first case of search list for item tests to see if the item is a list, and if so, calls the second case. The second case recursively calls search list for item. The recursive call has a Fail on Failure control so that if this call to search list for item fails by finding the search item in the list to be searched, this control passes the failure message upward to its calling method. (Other variations of these control flow constructs are also possible, but they do not impact exception handling.)

The failure exception and the controls described above constitute the exception handling mechanism of Prograph.
**Built-in exceptions:**

The failure exception is the only exception allowed, and is built-in. However, the only built-in primitive operations that signal the failure exception are the boolean primitives, and the **Match** operator. The Match operator signals failure if the input value is not equal to the value specified in it. (Other failures of built-in primitive operations, such as division by zero, do not raise the fail exception, but rather terminate the program.) For example, in Figure 24, if the input value is not 20, the **Match** operator signals a failure.

![Figure 24: Signaling exceptions using Match operator in Prograph.](image)

A boolean primitive signals a failure if it has no output nodes, and the condition it is testing evaluates to False. If an output node is specified, the operation returns True or False as output. In the example shown in Figure 25, if input value is not less than 10, the **<** operator evaluates to False, and control is immediately transferred to the calling function.
User-defined exceptions:

There are no user-defined exceptions. As mentioned before, there is only one possible exception in Prograph - the Fail exception, that can be signaled by the program code.

Continuation:

Raising a Fail exception causes the current case to immediately terminate. So, resume continuation is not possible, and only terminate continuation is available in Prograph.

Transferring information from signaling point to handler:

It is not possible to transfer information from the signaling point to the handler. The calling function only gets a fail message, with no additional information.
Default handlers:

There are no default handlers. If no handler is specified to catch the exception, the program is terminated.

Asynchronous events:

Asynchronous events cannot be handled by the fail/control mechanism. There is a separate application framework mechanism, with dataflow taps into the Macintosh toolbox, to deal with user interaction.

Grouping of exceptions:

Grouping of exceptions of either type is not available in Prograph, since there is only one possible exception.

Enforcement:

There is no compile time enforcement of the exception handling. The compiler does not do a static check to ensure that all fail exceptions are handled. Any unhandled fail message at run time causes an error to be generated, and the program terminates.

Other comments:

It is interesting to note that file system operations in Prograph do not rely on this exception handling mechanism to handle exceptions. Instead, each file operation returns an error value along with other output. The value returned is non-zero if there was an error. The language specifies a set of errors corresponding to each number. For example, Macintosh toolkit errors generate negative values, whereas program primitive errors
generate positive integers. The operations calling a file operation must then check for this error value to make that the operation succeeded, and take appropriate action if it didn’t.

Another interesting feature is the error state. Any unexpected event like a division by zero, or an type error causes the program to be in error state. There is no way for the program to handle this, and if a compiled program generates an error, it is terminated. However, if the program is being run in the Prograph interpreter, an error causes the debugger to be launched, with the offending code highlighted. The programmer can then interactively modify the code, and the program can restart execution from the point it stopped. This is very useful during development, and is an important feature of the Prograph development environment. However, it does not constitute exception handling, because the errors cannot be handled by the program itself.

**Haskell**

Haskell [Peterson et al. 1997a] is a general purpose, purely functional programming language that provides support for features like higher-order functions and static polymorphic typing. It is representative of recent functional languages.

Haskell makes a distinction between errors and exceptions. The term error is used for a condition that cannot be recovered from, such as non-termination or pattern match failure. The term exception, on the other hand, applies only to errors related to I/O that can be caught and handled [Hudak et al. 1997].

IO exception handling in Haskell is possible only within the IO monad. The IO monad can be considered to be an abstract data type inside which IO actions can be embedded, and a sequential ordering applied to them. A monad is a relatively recent device that solves the problem that normally, in a functional language, the order of evaluation of expressions is constrained only by data dependencies. IO and some other actions, on the other hand, need to be ordered in a well-defined manner to be meaningful.
The IO monad allows IO actions to be sequentially ordered in Haskell while still maintaining the purely functional nature of the language.

IO exceptions in Haskell are caught using the catch function. The catch function associates an exception handler with an action or set of actions. The type specification of the catch function is as follows:

\[
\text{catch} \quad :: \quad \text{IO} \ a \rightarrow (\text{I0Error} \rightarrow \text{IO} \ a) \rightarrow \text{IO} \ a
\]

The arguments to catch are an action and a handler. If the action succeeds, its result is returned without invoking the handler. If an error occurs, it is passed to the handler as a value of type I0Error and the handler is then invoked.

(The first term on the right hand of "::" is the type of the input to the function -an IO action in this case. The term in the parenthesis specifies the type specification of the exception handler to be invoked in case of an exception. The type specification of the handler in turn, indicates that it is given an I0Error as input and it returns an IO action as the output. The final term on the right hand side is the type of the value returned by the catch function).

**Built-in exceptions:**

Built-in exceptions are provided by the IO-monad in Haskell. All IO exceptions are of the abstract type I0Error. The built-in exceptions can only be signaled by the system primitives, not by user-written code. However, the program can query the value of an exception using some built-in functions. For example, the function:

\[
\text{isEOFError} \quad :: \quad \text{I0Error} \rightarrow \text{Bool}
\]

determines whether an exception was caused by an end-of-file condition.

Once an exception has been signaled, it can also be explicitly propagated using the fail command:

\[
\text{fail} \quad :: \quad \text{I0Error} \rightarrow \text{IO} \ a
\]
The fail command causes the exception to be signaled again, and this exception can then be caught and handled in the enclosing scope. In the following example, [Hudak et al. 1997] the function `getChar'` returns a character from the standard input. In case an exception is signaled by the function `getChar`, it is caught and passed on to the handler function `eofHandler`. If the exception was an end of file exception, the function returns a newline character. Otherwise, the exception is signaled again.

```haskell
getChar' :: IO Char
getChar' = getChar 'catch' eofHandler where
  eofHandler e = if isEOFError e then return '\n'
                 else fail e
```

**User-defined exceptions:**

User-defined exceptions can be created using the function `userError`:

```haskell
userError :: String -> IOError
```

The function `userError` takes a string as argument, and returns a value of type `IOError`. The program can then signal this exception using the fail function.

**Continuation:**

Only termination continuation is possible in Haskell, as raising of an exception causes the current function to terminate, and control to be transferred to a corresponding exception handler in the enclosing scope. However, it is possible for the handler to recursively retry the function that signaled the exception. In the following example, the function `getInputFile` prompts the user for a file name and then returns the contents of that file. In case it is unable to read the file, the exception handler gets invoked, which prints out an error message, and reinvokes `getInputFile`.

```haskell
getInputFile = do
  putStr "Copy from: 
  inFileName <- getLine
  catch (readFile inFileName)
  (\_ -> do
    putStr ("Can't open" ++ inFileName ++ "\n")
```
Transferring information from signaling point to handler:

Haskell exception handling provides a limited ability to transfer information from the signaler to the handler. The exceptions signaled by the built-in functions have information that indicates the name and file handle of the file whose manipulation caused the exception to be signaled. They also contain an error string. This information can be accessed by using the functions ioeGetErrorString, ioeGetHandle and ioeGetFileName. However, in user-defined exceptions, only the error string can be specified by the user.

Default handlers:

There are no default handlers for exceptions. If no handler is specified for an exception that gets signaled at run time, the program is terminated.

Asynchronous events:

It is not possible to handle asynchronous events with the exception handling mechanism as it is designed to work only for file IO within the IO monad.

Grouping of exceptions:

Grouping of exceptions of either kind is not available. The only exceptions allowed are the distinct built-in exceptions, and the one type of user-defined exception. All of these exceptions are handled by a handler which catches an exception of any kind thrown within its scope. So, there is no mechanism by which a subset of these exceptions could be collected in a group of different types, or different instances of a particular type.
Enforcement:

There is no compile time enforcement of the exception handling. The compiler does not do a static check to ensure that all exceptions are handled. Any unhandled exceptions at run time cause the program to be terminated.

Overall comparison

The chart in Table 4 shows a summary of the expressive power criteria satisfied by each language that we considered.

<table>
<thead>
<tr>
<th></th>
<th>Forms/3</th>
<th>MS Excel</th>
<th>Java</th>
<th>C++</th>
<th>Prograph</th>
<th>Haskell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Built-in exceptions</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>User-defined exceptions</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Terminate continuation</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Resume continuation</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transfer of information</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Default handlers</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asynchronous events</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Grouping of exceptions</td>
<td>Some</td>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enforcement</td>
<td></td>
<td></td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Expressive power of different exception handling mechanisms.
Conclusions

Exception handling is an important step towards making programs more robust and reliable. A well implemented exception handling mechanism can provide a program with the ability to recover from unexpected situations and errors. Even when it is impossible to recover, exception handling enables the program to execute some wrap-up action and exit gracefully.

In addition (as Forms/3 demonstrates) exception handling can provide a unified mechanism for handling interactive events (such as mouse clicks), hardware errors (e.g. division by zero), user-generated errors (e.g. user inputs of the wrong type), and system errors (like running out of stack space). This could make a language easier to use, by reducing the number of constructs programmers have to learn.

In this work we have presented the following contributions:

- We improved the exception handling in Forms/3 to ensure that it completely upholds the property of referential transparency and laziness in the presence of exceptions. In particular we ensured that the operators and, or and if are properly lazy even in the presence of exceptions.

- We demonstrated that exception handling in Forms/3 can deal with multiple exceptions without violating principles of referential transparency.

- We demonstrated that the expressive power of Forms/3 exception handing mechanism compares very favorably to mechanisms available in other conventional languages.
References


