1.6. Tratamento de Exceções

Uma exceção denota um comportamento anormal, indesejado, que ocorre raramente e requer alguma ação imediata em uma parte do programa [GHE 97, DER 90]. Esta condição pode ser um erro, tal como um *overflow* aritmético, ou pode ser uma ocorrência anormal que não é considerada um erro, tal como o fim do arquivo. Condições que são consideradas exceções são chamadas de síncronas, o que significa que elas ocorrem em locais previsíveis do programa. Por exemplo, o *overflow* ocorrerá somente onde está sendo feita uma operação aritmética, e o fim do arquivo ocorrerá somente quando um arquivo estiver sendo lido. Isto distingue exceções de condições que são assíncronas, isto é, que podem ocorrer a qualquer momento. Uma interrupção gerada pelo usuário ou um sinal do dispositivo são exemplos de condições assíncronas, que são gerenciadas mais apropriadamente através das características de concorrência de uma LP, uma vez que são geradas através de algum processo que está executando concorrentemente [DER 90].

Exceções possuem um sentido mais amplo do que simplesmente erros computacionais. Elas referem-se a qualquer tipo de comportamento anômalo que, intuitivamente e informalmente, correspondem a um desvio do curso de ações esperado, previsto pelo programador. O conceito de "desvio" não pode ser colocado rigorosamente. Ele representa uma decisão de projeto tomada pelo programador, que decide que certos estados são "normais" e "esperados", enquanto outros são "anormais". Assim, a ocorrência de uma exceção não significa necessariamente que ocorreu um erro catastrófico, mas sim que a unidade que está sendo executada não pode proceder da maneira definida pelo programador [GHE 97].

Normalmente, os procedimentos são invocados através de uma chamada explícita, que coloca o procedimento em execução e suspende a execução da unidade que o chamou (figura 1.9). Após a execução completa do procedimento, a unidade que o chamou volta a ser executada a partir do ponto da invocação. Procedimentos que são invocados implicitamente através da ocorrência de uma exceção são chamados de tratadores de exceções. Assim como na chamada de um procedimento, o tratamento de exceções resulta na suspensão da execução da unidade que o invocou. Porém, duas ações diferentes são possíveis no término do tratamento de exceções: continuação da execução da unidade que fez a invocação, assim como ocorre com procedimentos, ou término da execução de tal unidade. Estas duas abordagens estão ilustradas na figura 1.10.

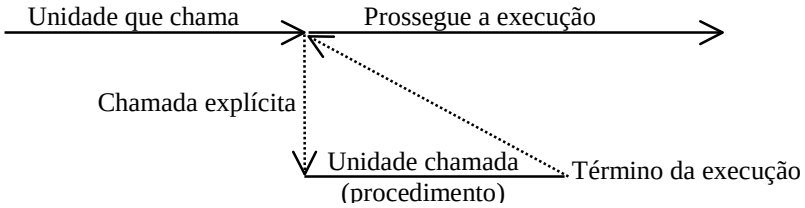

Figura 1.9 – Execução de procedimentos [DER 90]

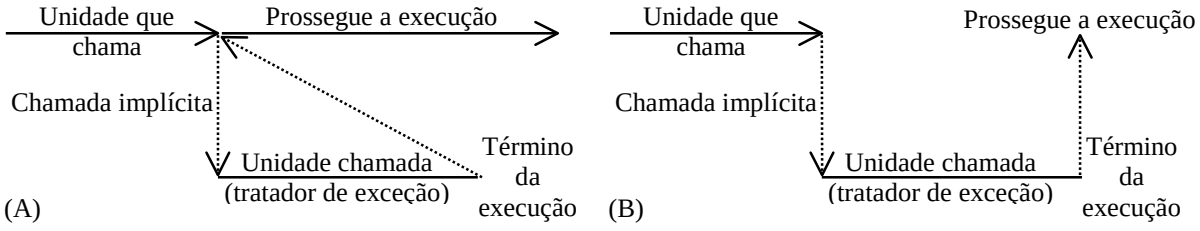

Figura 1.10 – Tratamento de exceções: (A) continuação da execução da unidade que chama, (B) término da execução da unidade que chama [DER 90]

A invocação implícita de uma exceção é geralmente referenciada como sinalização da exceção. A maioria das linguagens, entretanto, fornece facilidades para invocação explícita de exceções, o que ocorre através de um comando de sinalização, que consiste, geralmente, em uma palavra chave tal como *RAISE* seguida do nome da

exceção. Isto permite que o programa sinalize uma exceção para invocar o tratador de exceções neste ponto, mesmo sem a ocorrência da condição associada. Além disso, a linguagem também pode permitir que o usuário defina exceções. Tais exceções podem ser declaradas como outro tipo de dado qualquer e irão existir dentro do escopo das suas declarações. Exceções definidas pelo usuário devem ser sinalizadas explicitamente, uma vez que não possuem condição associada. Elas comportam-se como procedimentos e podem até aceitar parâmetros. A maior diferença está no fluxo do controle no término da unidade invocada. Procedimento sempre irão retornar ao ponto imediatamente após a invocação, enquanto um tratamento de exceções invocado explicitamente pode proceder de maneira diferente. Uma linguagem deve permitir a habilitação ou não de exceções, e cabe ao programador decidir se uma exceção deve ser habilitada em tempo de compilação.

Tratadores de exceções consistem em blocos de comandos que são amarrados a uma exceção (figura 1.11). Para exceções fornecidas pela linguagem a amarração do nome é permanente. Exceções definidas pelo usuário são amarradas aos seus nomes no ponto de declaração, e esta amarração permanece dentro do escopo da declaração [DER 90].

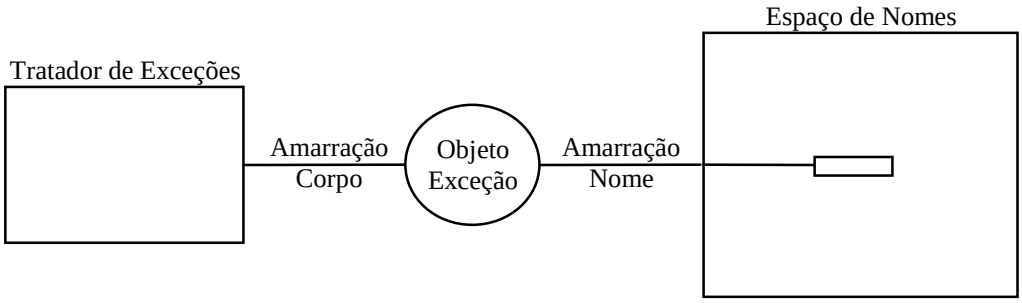

Figura 1.11 – Objeto exceção [DER 90]

Para definir o tratamento de exceções, as seguintes decisões devem ser tomadas pelo projetista da LP:

1. Quais são as exceções que devem ser tratadas? Como elas podem ser definidas?
2. Que unidades podem sinalizar uma exceção e como?
3. Como e onde um tratador pode ser definido?
4. Como é a amarração entre uma exceção e o seu tratador e como o controle flui de uma exceção sinalizada para o seu gerenciador?
5. Para onde o controle flui depois de uma exceção ter sido tratada?

As respostas fornecidas para tais questões, que podem ser diferentes em cada linguagem, afetam a semântica do tratamento de exceções, sua utilidade e sua facilidade de implementação. Agora as soluções fornecidas por algumas linguagens (*Ada*, *C++*, *Java* e *ML*) para o tratamento de exceções são apresentadas [GHE 97].

Ada fornece um conjunto de quatro exceções pré-definidas que podem ser automaticamente detectadas e sinalizadas em tempo de execução:

- *Constraint_Error*: verificação da violação de um limite, tais como o índice de um vetor e a divisão por zero.
- *Program_Error*: verificação de uma falha de uma regra da linguagem. Por exemplo, espera-se que uma função termine normalmente através da chamada de um comando de retorno, e isto não acontece.
- *Storage_Error*: verificação de falta de memória quando é feita uma alocação dinâmica.
- *Tasking_Error*: verificação da ocorrência de um erro durante a execução concorrente de tarefas.

Além destas exceções, um programa também pode declarar novas exceções, tal como:

Help: ***exception****;*

Uma unidade de programa pode, então, sinalizar explicitamente a exceção "*Help*" com:
***raise** Help;*

Uma vez sinalizadas, exceções pré-definidas e definidas pelo usuário comportam-se exatamente da mesma maneira. Tratadores de exceções podem ser vinculados ao corpo de um subprograma, ao corpo de um pacote ou a um bloco, depois da palavra chave *exception*. Por exemplo:

```
begin -- Este é um bloco com tratadores de exceção
    ...comandos...
   exception when Help=>tratamento da exceção Help;
             when Constraint_Error=>tratamento da exceção pré-definida Constraint_Error;
             when others=>tratamento para todas as outras exceções
end;
```

Neste exemplo, uma lista de tratadores é vinculada ao bloco. A lista é definida através da palavra chave *exception* e cada tratador através de *when*.

Se a unidade que sinalizou a exceção fornece um tratador para ela, o controle é imediatamente transferido para aquele tratador: as ações que seguem o ponto onde a exceção foi acusada não são executadas, o tratador é executado, e então o programa continua a execução como se a unidade que sinalizou a exceção fosse finalizada normalmente, isto é, a partir do ponto depois do término do bloco. Entretanto, se a unidade que está executando não fornece um tratador, a unidade termina e a exceção é propagada. Propagação significa que a exceção é sinalizada novamente em outro contexto. Em geral, se a propagação da exceção não é tratada no ponto para onde ela foi transferida, ela é propagada além disso, e a propagação pode eventualmente levar ao término do programa. O efeito preciso da propagação depende do tipo de unidade que sinalizou a exceção, se é um bloco, um procedimento, um corpo de pacote ou um corpo de tarefa [GHE 97].

Em ***C++***, exceções podem ser geradas através da execução em tempo real, como por exemplo numa divisão por zero, ou podem ser sinalizadas explicitamente pelo programa. Uma exceção é sinalizada pela instrução *throw*, que transfere um objeto para o tratador correspondente. Um tratador pode ser vinculado a qualquer pedaço de código (um bloco) que necessite do tratamento de falhas. Para fazer isto, o bloco deve ser pré-definido pela palavra chave *try*. Considere o seguinte caso:

```
class Help {...};
//objetos desta classe possuem um atributo público "kind" do tipo enumeração descrevendo o
// tipo de ajuda requisitada, e outros campos públicos que podem carregar informação
// específica sobre o ponto no programa onde ajuda é requisitada
class Zerodivide {};
// assume que os objetos desta classe são gerados através do tempo de execução do sistema
...
try {
    // bloco de tolerância a falhas que pode sinalizar as exceções "Help" ou "Zerodivide"
    ...
}
catch (Help msg) {
    // trata o pedido de ajuda transmitido pela mensagem do objeto
    switch (msg.kind) {
       case MSG1: .......;
       case MSG2: .......;
    }
}
catch (Zerodivide) {  // trata a divisão por zero
}
```

Suponha que o bloco *try* contém o comando

throw *Help(MSG1);*

A expressão *throw* faz com que a execução do bloco seja abandonada e o controle é transferido para o tratador apropriado. No exemplo, "*Help(MSG1)*" invoca o construtor da classe "*Help*" passando um parâmetro que é usado pelo construtor para inicializar o campo "*kind*". O objeto temporário criado é usado para inicializar o parâmetro formal "*msg*" do *catch* correspondente, e o controle é então transferido para a primeira opção ("*case MSG1*") do comando *switch* no primeiro tratador vinculado ao bloco.

Resumindo: se uma exceção ocorrer dentro do bloco *try* (bloco de prova), será disparada usando *throw*; a exceção deve ser pega por um comando *catch* que segue imediatamente o comando *try* que dispara a exceção, isto é, se uma exceção é chamada no bloco *try* e o controle do programa é transferido para o gerenciador de exceção apropriado. Pode haver mais de um comando *catch* associado com um *try*. De acordo com o tipo da exceção é determinado qual comando *catch* será usado.

Caso não seja fornecido um tratador para a exceção, a execução da rotina é abandonada e a exceção é propagada para o ponto da chamada dentro do bloco. A execução do bloco é, por sua vez, abandonada e o controle é transferido para o tratador como no caso anterior. Em outras palavras, *C++*, assim como *Ada*, propaga exceções não tratadas. Também como em *Ada*, a exceção pode ser propagada explicitamente através de *throw*, e depois do tratador ser executado, a execução continua como se a unidade à qual o tratador é vinculado tivesse completado normalmente. Ao contrário de *Ada*, entretanto, qualquer quantidade de informação pode ser transmitida com uma exceção. Para sinalizar uma exceção é possível fazer o *throw* de um objeto que contém dados que podem ser usados pelo tratador.

Como em *C++*, exceções ***Java*** são objetos que podem ser invocados e capturados por tratadores vinculados aos blocos. As principais diferenças introduzidas por *Java* são:

1. Todas as exceções que podem ser invocadas explicitamente através de uma rotina devem ser listadas na interface da rotina, como por exemplo: *void foo() throws Help;*
2. Um bloco tolerante a falhas que pode tratar exceções é chamado bloco *try*, e tem a forma apresentada a seguir. Neste caso, a seção "bloco_final", se presente, realiza qualquer ação de finalização necessária.

```
try
    block;
catch (exceção_tipo_1)
    tratador_1;
    ...
catch (exceção_tipo_n)
    tratador_n
finally
    bloco_final
```

A linguagem funcional ***ML*** permite que exceções sejam definidas, sinalizadas e tratadas. Existem também exceções que são pré-definidas pela linguagem e sinalizadas automaticamente durante o tempo de execução. O próximo exemplo mostra a declaração de uma exceção,

exception Neg

que pode ser sinalizada subseqüentemente na seguinte declaração de função:

```
fun fact(n)
    if n < 0 then raise Neg
    else if n = 0 then 1
        else n * fact (n-1)
```

Uma chamada, tal como *fact(-2)* pode fazer com que a avaliação de uma função seja abandonada e a exceção sinalizada, e, desde que nenhum tratador é fornecido, o programa irá parar escrevendo a mensagem "*Failure: Neg*".

Supondo que se queira tratar a exceção retornando zero quando a função é chamada com um argumento negativo. Isto pode ser feito, por exemplo, através da definição da seguinte nova função:

```
fun fact_0(n) =
    fact (n) handle Neg => 0;
```

que usa "*fact*" como uma função auxiliar.

Exceções que não são tratadas em uma cadeia de chamadas de funções são implicitamente propagadas. Isto é, supondo-se que a função "*fact*" é chamada através de alguma função "*f*" que não fornece um tratador para "*Neg*"; a função "*f*" é, por sua vez, chamada pela função "*g*", que fornece um tratador para "*Neg*" da mesma maneira que a função "*fact_0*" faz. Neste caso, a avaliação da expressão

g (f(fact(-33)))

leva a zero [GHE 97].