Ludwig Maximilians Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Skript zur Vorlesung
Informatik I
Wintersemester 2006

# Kapitel 10: Auswertung und Ausnahmen: Der SML – Typ exn („exception“)

Vorlesung: Prof. Dr. Christian Böhm
Übungen: Elke Achtert, Arthur Zimek



http://www.dbs.ifi.lmu.de/Lehre/Info1

Database Systems

## Der vordefinierte Typ `exn`

- SML bietet den vordefinierten Typ `exn` (*exception* oder *Ausnahme*) an.
- Seine Werte heißen Ausnahmewerte (*exception values*).
- Besonderheit:

  *Der Programmierer kann dem Typ neue (Wert-) Konstruktoren hinzufügen.*
- Dies ist für keinen vordefinierten Typ, und auch für keinen vom Programmierer definierten Typ, möglich.
- Die neuen (Wert-) Konstruktoren des vordefinierten Typs `exn`, die in einem Programm deklariert werden können, werden Ausnahmekonstruktoren (*exception constructors*) genannt.

# Ausnahmekonstruktoren

- Ein Ausnahmekonstruktor namens `A` wird so deklariert:

  ```
  exception A;
  ```

- Ausnahmekonstruktoren können konstant sein:

  ```
  exception illegal_expression;
  ```

- Ausnahmekonstruktoren können aber auch einen Parameter von irgend einem Typ `t` haben.
- Ein Ausnahmekonstruktor namens `A` mit Parameter vom Typ `t` wird deklariert:

  ```
  exception A of t;
  ```

- Ausnahmen können unter Verwendung des `let` - Konstrukts von SML auch lokal deklariert werden.



# Ausnahmen erheben (oder werfen)

- Eine Ausnahme `A` wird mit dem Ausdruck:

  ```
  raise A
  ```

  erhoben (oder geworfen).

# Beispiel: Erhebung einer konstanten Ausnahme

```
- exception negative_integer;
exception negative_integer

- fun factorial x =
            if x < 0
            then raise negative_integer
            else if x = 0
                 then 1
                 else x * factorial(x - 1);
val factorial = fn : int -> int

- factorial 4;
val it = 24 : int

- factorial ~4;
uncaught exception negative_integer
```



# Beispiel: Erhebung einer Ausnahme mit Parametern

```
- exception negative_argument of int;
exception negative_argument of int

- fun fac x =
            if x < 0
            then raise negative_argument(x)
            else if x = 0
                 then 1
                 else x * fac(x - 1);
val fac = fn : int -> int

- fac ~4;
uncaught exception negative_argument
```

Die Ausnahme hat den Ausnahmeparameter ~4, der aber in der gedruckten Mitteilung des SML-Systems nicht erwähnt wird. Wie in vielen Fällen verkürzt SML/NJ die gedruckte Mitteilung, so dass die eigentliche Struktur nicht vollständig sichtbar wird.

# Ausnahmen als Werte

- Ausnahmekonstruktoren werden weitgehend wie herkömmliche (Wert-) Konstruktoren verwendet (vgl. Kapitel 5 und 8).
- Ausnahmekonstruktoren können u.a. zum Aufbau von neuen Werten und zum Musterangleich verwendet werden.
- Beispiel:

  Listen von Ausnahmewerten können so gebildet werden.
- Ausnahmen können unter Verwendung des `let` - Konstrukts von SML auch lokal, auch zu der Definition einer rekursiven Funktion, definiert werden.



# Beispiel

```
- fun f x = let
              exception invalid_argument
            in
              if x < 0
              then raise invalid_argument
              else if x = 0
                   then 1
                   else x * f(x - 1)
            end;
val f = fn : int -> int

- f ~4;
uncaught exception invalid_argument
```

# Merke

- Bei lokalen Deklarationen von Ausnahmen können unterschiedliche Ausnahmen denselben Namen tragen.
- Das kann zu Verständnisproblemen führen.
- Dies sollte vermieden werden, und Ausnahmen sollten so weit wie möglich nur global deklariert werden.



# Ausnahme versus Wert (1)

- Wie Werte von herkömmlichen Typen können Ausnahmen das Ergebnis einer Auswertung sein.
- Beispiel:
  Wenn die Fakultätsfunktion `factorial` auf eine negative ganze Zahl angewandt wird:

```
- factorial ~4;
uncaught exception negative_integer
```

# Ausnahme versus Wert (2)

- Ausnahmen werden während der Auswertung nicht wie Werte von herkömmlichen Typen behandelt, sondern wie folgt:

  Liefert die Auswertung eines Teilausdruckes `T` eines zusammengesetzten Ausdrucks `B` eine Ausnahme `A` als Ergebnis, so wird diese Ausnahme `A` als Ergebnis der Auswertung des Gesamtausdrucks `B` geliefert, es sei denn, `A` wird von einem Ausnahmebehandler eingefangen.



# Beispiel: Sonderstellung von Ausnahmen während der Auswertung

```
- exception negative_integer;
exception negative_integer
- fun factorial x = if x < 0
                    then raise negative_integer
                    else if x = 0
                         then 1
                         else x * factorial(x - 1);
val factorial = fn : int -> int
- fun is_even x = let val x_mod_2 = x mod 2
                  in
                    case x_mod_2
                    of 0   => true
                     | _   => false
                  end;
val is_even = fn : int -> bool
- is_even(factorial ~4);
uncaught exception negative_integer
```

Nach der Auswertung in applikativer Reihenfolge führt die Auswertung von: `is_even(factorial ~4)` zunächst zur Auswertung von `factorial ~4`. Diese Auswertung liefert als Ergebnis die Ausnahme `negative_integer`.

# Ausnahmebehandlung

- Würden Ausnahmewerte bei der Auswertung wie herkömmliche Werte behandelt, so müsste anschließend `is_even(negative_integer)` ausgewertet werden.
- Das könnte auf zwei Weisen geschehen:
  - Ein Typfehler könnte zur Laufzeit gemeldet werden, weil `is_even` einen Aufrufparameter vom Typ `int`, aber nicht `exn` erwartet.
  - Der Ausnahmewert könnte wie ein Sonderwert verarbeitet werden, so dass (`negative_integer mod 2`) wiederum den Wert `negative_integer` liefert. Dann würde die Auswertung von `is_even(negative_integer)` wegen des Fangfalls im `case` - Konstrukt den Wert `false` liefern.
- Die Auswertung von `is_even(factorial ~4)` ergibt aber den Ausnahmewert `negative_integer`. → Der Ausnahmewert wurde bei der Auswertung NICHT wie ein herkömmlicher Wert behandelt.



# Ausnahmen behandeln (oder einfangen) (1)

- Ein Ausnahmebehandler (*exception handler* ) ist eine Art Funktion, die aber nur Parameter vom Typ `exn` haben kann.
- Er hat im einfachsten Fall die Gestalt: `handle A => C`
- Ein Ausdruck `B` kann mit einem Ausnahmebehandler verknüpft werden: `B handle A => C`
- Liefert die Auswertung von `B` (oder einem Teilausdruck `T` in `B`) die Ausnahme `A`, so wird der Ausnahmebehandler `handle A => C` wirksam:
  - Ausdruck `C` wird ausgewertet und der Wert von `C` als Wert von `B` geliefert.
  - Die Ausnahme `A` wird also nicht weitergereicht.
- Liefert die Auswertung von `B` etwas anderes als die Ausnahme `A`, so hat der Ausnahmebehandler `handle A => C` keine Wirkung.

# Ausnahmen behandeln (oder einfangen) (2)

- Eine häufige Veranschaulichung besteht in der Vorstellung, dass ein Teilausdruck `T` tief innerhalb von `B` die Ausnahme `A` in Richtung seiner umfassenden Ausdrücke wirft.
- Die Ausnahme `A` wird von den umfassenden Ausdrücken von `T` einfach durchgelassen, bis sie bei `B` vom Ausnahmebehandler eingefangen wird.



# Beispiel: Ergänzung zu `factorial` (1)

```
- exception negative_integer;
exception negative_integer

- fun factorial x =
            (if x < 0
             then raise negative_integer
             else if x = 0
                  then 1
                  else x * factorial(x - 1))
             handle negative_integer => factorial(~x);
val factorial = fn : int -> int

- factorial ~4;
val it = 24 : int
```

# Beispiel: Ergänzung zu `factorial` (2)

- Wird die Funktion `factorial` auf eine negative ganze Zahl `x` angewandt, so wird die Ausnahme `negative_integer` erhoben.
- Diese Ausnahme wird vom Behandler

  `handle negative_integer => factorial(~x)`

  eingefangen, was zur Auswertung von

  `factorial(~(~4))`

  führt.



# Beispiel: Behandler für `negative_argument` in `fac`

```
- exception negative_argument of int;
exception negative_argument of int

- fun fac x = if x < 0
              then raise negative_argument(x)
              else if x = 1
                   then 1
                   else x * fac(x - 1);
val fac = fn : int -> int

- fac ~4 handle negative_argument(y) => fac(~y);
val it = 24 : int
```

# Vergleich der beiden Beispiele

- Obwohl in beiden Beispielen die Anwendung der Fakultätsfunktion auf eine negative ganze Zahl `z` zur Berechnung der Fakultät des Betrags `~z` dieser Zahl führt, geschieht dies jeweils auf unterschiedliche Weise:
  - Die Ausnahme, die erhoben wird, wird im ersten Beispiel innerhalb des Rumpfes der Fakultätsfunktion `factorial` eingefangen, d.h. im Geltungsbereich des formalen Parameters `x` der Funktion `factorial`.
  - Im zweiten Beispiel ist der Behandler außerhalb des Geltungsbereiches des formalen Parameters `x` der Fakultätsfunktion `fac` definiert.

    Der Wert `~4` des aktuellen Parameters des Aufrufes wird über die einstellige Ausnahme `negative_argument` an den Behandler `negative_argument(y) => fac(~y)` weitergereicht.



# Allgemeine Form einer Behandler – Definition

- Die allgemeine Form der Definition eines Behandlers ist:

```
handle <Muster1> => <Ausdruck1>
  |     <Muster2> => <Ausdruck2>
                  .
                  .
                  .
  |     <Mustern> => <Ausdruckn>
```

- Ein Behandler besteht also aus einem Angleichmodell.

# Achtung bei Schreibfehlern!

- Beispiel:

  Kommt statt eines Ausnahmenamens

  `illegal_expression`

  in einem Muster fälschlich ein Bezeichner wie

  `ilegal_expression`

  vor, so wird der Bezeichner als ungebundener Name (oder Variable) ausgelegt, der während des Musterangleichs an jede mögliche Ausnahme gebunden werden kann.



# Prinzip der Auswertung von `raise`- und `handle`- Ausdrücken

- Die Auswertung eines Ausdrucks:

  `raise <Ausdruck>`

  führt zur Auswertung von `<Ausdruck>`, der ein Ausdruck vom Typ `exn` sein muss.
- Liefert die Auswertung von `<Ausdruck>` einen Ausnahmewert `<Ausdruck>`, so ist der Wert des Ausdrucks:

  `raise <Ausdruck>`

  das sogenannte Ausnahmepaket `$[<Ausnahme>]`.
- Der Begriff Ausnahmepaket dient hauptsächlich dazu, Ausnahmewerte von herkömmlichen Werten zu unterscheiden.

`$[<Ausnahme>]` ist KEINE SML – Notation und stellt insbesondere keine Liste dar!

# Ausnahmepaket als Ergebnis der Auswertung

- Bei der Auswertung eines zusammengesetzten Ausdrucks werden zunächst die Teilausdrücke ausgewertet.
- Ist einer der dabei ermittelten Werte ein Ausnahmepaket, wird die weitere Auswertung der Teilausdrücke abgebrochen und das Ausnahmepaket als Wert geliefert.
- Auf diese Weise wird das Ausnahmepaket als Ergebnis der Auswertungen sämtlicher umfassender Ausdrücke weitergereicht, bis ein Behandler gefunden wird.



# Allgemeines Beispiel

- Bei der Auswertung eines Ausdrucks
  `<Ausdruck1> handle <Muster> => <Ausdruck2>`
  wird zunächst `<Ausdruck1>` ausgewertet:
  - Ist der Wert KEIN Ausnahmepaket, wird dieser Wert geliefert.
  - Ist der Wert ein Ausnahmepaket `$[<Ausnahme>]`, erfolgt ein Musterangleich zwischen:
    `<Muster>`
    und dem Ausnahmewert:
    `<Ausnahme>`.
- Bei Erfolg wird `<Ausdruck2>` ausgewertet und dessen Wert geliefert.
- Bei Misserfolg wird das Ausnahmepaket `$[<Ausnahme>]` geliefert.

# Beispiel: Auswertung von `raise-` und `handle-` Ausdrücken (1)

```
- exception negative_zahl;

- fun f x = if x = 0
            then true
            else if x > 0
                 then f(x - 1)
                 else raise negative_zahl;
val f = fn : int -> bool

- fun g true = "wahr"
  | g false = "falsch";
val g = fn : bool -> string

- g(f ~3) handle negative_zahl => "Fehler";
val it = "Fehler" : string
```



# Erläuterung durch das Substitutionsmodell (1)

```
g(f ~3) handle negative_zahl => "Fehler"

g(if ~3 = 0
  then true
  else if ~3 > 0
       then f(~3 - 1)
       else raise negative_zahl ) handle negative_zahl => "Fehler"

g(if false
  then true
  else if ~3 > 0
       then f(~3 - 1)
       else raise negative_zahl ) handle negative_zahl => "Fehler"
```

# Erläuterung durch das Substitutionsmodell (2)

```
g(    if ~3 > 0
      then f(~3 - 1)
      else raise negative_zahl ) handle negative_zahl => "Fehler"

g(    if false
      then f(~3 - 1)
      else raise negative_zahl ) handle negative_zahl => "Fehler"

g( raise negative_zahl ) handle negative_zahl => "Fehler"

g( $[negative_zahl] ) handle negative_zahl => "Fehler"

$[negative_zahl] handle negative_zahl => "Fehler"

"Fehler"
```

- Erinnerung: `$[negative_zahl]` ist ein Ausnahmepaket.



# Ausnahmen und Auswertungsreihenfolge (1)

- Ausnahmen sind ihrer Natur nach eng an die Auswertungsreihenfolge gekoppelt.
- Mit SML - Ausnahmen kann man herausfinden, in welcher Reihenfolge Teilausdrücke ausgewertet werden:

```
- exception a;
exception a
- exception b;
exception b
```

Wäre zuerst der zweite Teilausdruck von + ausgewertet worden, so wäre das Gesamtergebnis 2 gewesen.

```
- ((raise a) + (raise b)) handle  b => 2
                                |     a => 1;
val it = 1 : int
```

# Ausnahmen und Auswertungsreihenfolge (2)

- Wie man eine Ausnahmebehandlung im Stil von `raise` und `handle` so in funktionale Sprachen integrieren kann, dass sie nicht so eng an die Auswertungsreihenfolge gekoppelt ist, ist derzeit Gegenstand der Forschung.



# Vordefinierte Ausnahmen von SML

- Einige Ausnahmen sind in SML vordefiniert:
  z.B. die Ausnahmen `Match` und `Bind`, die bei Fehlern während des Pattern Matching erhoben werden.
- Die Standardbibliothek von SML beschreibt die vordefinierten Ausnahmen (vgl. Kapitel 2.10 und „The Standard ML Basis Library“ unter: http://www.smlnj.org/doc/basis/).