
Hallo, Ihr da draußen, hier ist der achte Teil von „Icon auf dem Raspberry Pi“. Hoffentlich konntet Ihr den siebten Teil genießen und die kleinen Listings erfolgreich compilieren und ausführen. Wenn ja, dann solltet Ihr die strukturierten Datentypen `list`, `record`, `table`, `set` bzw. `cset` verstanden haben und anwenden können.

Lösung der Übungsaufgabe:
home/pi/icon9_51/bin/challenge solution7.icn

```
1  procedure main()
2     ref := table()
3     spaces := []
4     files := []
5
6     linenumber := 0
7     filename := "/home/pi/.config/geany/filedefs/filetypes.Icon.conf"
8     fh := open(filename, "r")
9     while read(fh) ~== "[keywords]"
10    kommandos := read(fh)
11    close(fh)
12
13    every put(spaces, find(" ", kommandos))
14    push(spaces, 8)
15    put(spaces, *kommandos)
16    every i := 1 to *spaces - 1 do ref[kommandos[spaces[i] + 1:spaces[i+1]]] := []
17
18    dir := open("dir /home/pi/icon9_51/bin/*.icn", "p")
19    if dir then
20    {  while datei := read(dir) do
21       {  fh := open(datei, "r")
22          if fh then
23          {  zeilennummer := 0
24             while zeile := read(fh) do
25             {  zeilennummer +:= 1
26                zeile ?
27                {  while tab(upto(&letters)) do
28                   {  word := tab(many(&letters))
29                      if member(ref, word) then
30                      {  if find(datei[23:0], ref[word][*ref[word]]) then ref[word][*ref[word]]
31 ||:= ", " || zeilennummer
32                         else put(ref[word], datei[23:0] || ":" || zeilennummer)
33                      }
34                   }
35             }
36          }
37          close(fh)
38       }
39       else stop("Datei ", datei, " kann nicht eingelesen werden!")
40    }
41    close(dir)
42 }
43 else stop("Zielverzeichnis-Fehler!")
44
45 every pair := !sort(ref) do
46 {  writes(left(pair[1], 20, "."))
47    every i := 1 to *pair[2] do
48    {  if i > 1 then writes(repl(" ", 20))
49       write(pair[2][i], " ")
50    }
51 }
52 end
```

Zunächst lesen wir die Zeile in `filetypes.Icon.conf`, die die Funktionen und Kommandos definiert (Zeilen 7 - 10). Die mit einem Leerzeichen getrenhten Worte werden in der Zeichenkette `kommandos` gespeichert. Wir ermitteln die Positionen der Leerzeichen der Zeichenkette `kommandos` und speichern dies in der Liste `spaces` ab (Zeile 13). Wir setzen zwei weitere Positionen, nämlich 8 (entspricht `primary=$define`, um `$define` als Befehl zu kennzeichnen) und die Länge der Zeichenkette `kommandos`, um den letzten Eintrag zu erkennen (Zeilen 14 - 15). Jeder durch Leerzeichen getrennte Befehl wird in die Tabelle `ref` als Schlüssel übernommen (Zeile 16). Wir öffnen eine Pipe (Zeile 18), die die Dateinamen der Icon-Dateien im Arbeitsverzeichnis `/home/pi/icon9_51/bin` ausliest. Die Dateinamen werden zeilenweise aus der Pipe ausgelesen (Zeile 20) und diese Dateien geöffnet (Zeile 21) und zeilenweise ausgelesen (Zeile 24). Die Zeileninhalte werden wie im zugrundeliegenden Programm untersucht (Zeilen 26 – 35). Bei Übereinstimmung eines Wortes des eingelesenen Quellcodes mit einem Befehl als Schüssel der Tabelle `ref` wird der Programmname zusammen mit der Zeilennummer in die Liste gesetzt (Zeile 32), die als Tabellenelement angelegt wurde (Zeile 16). Wenn dieser Befehl in einer

weiteren Zeile dieses Quellcodes vorkommt, wird nur noch die Zeilennummer eingetragen (Zeile 31).
Nach Ablauf dieses Programmes haben wir ein Dokument, das fast alle Icon- Befehle in einer lauffähigen Umgebung enthält. Wenn Ihr die Befehle in `filetypes.Icon.conf` regelmäßig auf den aktuellen Stand bringt, und dann gegen Ende dieses Tutorials dieses Programm erneut durchführt, dann habt Ihr ein Verzeichnis aller Icon-Befehle und Schlüsselworte inkl. Referenzen auf deren Anwendung in einer lauffähigen Umgebung!

# 8 Die Datentypen procedure und co-expression

In Icon kennen wir 14 Datentypen:

- integer, real (Teil 5), char, string, null (Teil 6)
- Strukturierte Datentypen: list, record, table, set, cset (letzte Ausgabe)
- file (Teil 3), window (Teil 15), **procedure** (diese Ausgabe)
- **co-expression** (diese Ausgabe)

## 8.1 Der Datentyp procedure

Da fast jede Prozedur den Zustand von Erfolg / Misserfolg zurückgibt, kann das Ergebnis einer Prozedur Bezeichnern zugewiesen werden, kann eine Prozedur als ein Argument zu anderen Prozeduren gesetzt werden (wie wir es schon mehrfach getan haben). Daher haben wir allen Grund zur Annahme, dass `procedure` ebenfalls ein Datentyp ist.

### 8.1.1 Deklaration

```
procedure
name([comma_separated_list_of_arguments])
   [static comma_separated_list_of_identifiers]
   [local comma_separated_list_of_identifiers]
   [initial {initial_clause}]
   expressions
end
```

Auf diese lokal angegebenen Bezeichner kann nur innerhalb dieser Prozedur zugegriffen werden – nirgendwo anders im Programm. Parameter, die in verschiedenen Prozeduren denselben Namen tragen, weisen keine Verbindung untereinander auf. Diese werden *lokale Bezeichner* genannt.

Die optionale `initial`-Anweisung wird nur beim allerersten Aufruf dieser Funktion durchgeführt.

Eine Prozedur kann deklariert werden, eine variable Anzahl von Argumenten zu haben. Du kannst das erreichen, indem Du an den letzten Parameter der Parameterliste ein [] anhängst: `procedure test(a,b,c,d[])` wird durch `test(1,2,3,4,5)` aufgerufen. Diese Liste besteht aus den Argumenten, die den bisherigen Parametern nicht zugewiesen wurden. Falls die Parameter innerhalb der Argumentenliste, die in der Prozedurdeklaration deklariert wurden, alle Parameter des Prozeduraufrufs verbrauchen, dann ist diese Liste `d[]` leer.

Den gleichen Effekt kannst Du durch `procedure test(a,b,c,d)` und Aufrufen von `test(1,2,3,[4,5])` erreichen.

### 8.1.2 Gültigkeitsbereich

#### *8.1.2.1 Globale Bezeichner*

Ein Bezeichner kann als global deklariert werden, um im gesamten Programm darauf zugreifen zu können und ohne dass dessen Wert von einer Prozedur zur anderen verloren geht.

```
global comma_separated_list_of_identifiers
```

ACHTUNG: Ein global deklarierter Bezeichner verliert seine Bedeutung, falls er in einer Prozedur lokal deklariert wird oder falls er in einer Parameterliste verwendet wird. Also: Lass das sein!

Jeder Prozedurname ist ein globaler Bezeichner: Du kannst jeden Prozedurnamen ohne irgendeine weitere Deklaration verwenden, sogar wenn externe Quelldateien oder Objektdateien (u-code files) verwendet werden.

Der Name von `record`-Deklaratrionen ist ebenfalls global. Record-Feldnamen sind keine Bezeichner!

#### *8.1.2.2 Statische Bezeichner*

Alle lokalen Bezeichner existieren nach Rückkehr der Funktion nicht mehr (oder wenn diese `end` erreicht). Wenn Du möchtest, dass die lokalen Bezeichner nach Rückkehr einer Prozedur weiterhin existieren, dann musst Du den Bezeichner als `static` deklarieren

```
static komma_getrennte_Bezeichnerliste
```

Das nächste Mal, wenn die Prozedur aufgerufen wird, behalten die als `static` deklarierten Bezeichner ihre Werte, die sie das letzte Mal in dieser Prozedur innehatten.

Vergiss nicht, einen Bezeichner als `static` zu deklarieren, falls er innerhalb einer `initial`-Anweisung steht! Deine Prozedur wird beim ersten Aufruf gut funktionieren, aber bem zweiten Mal ist der Bezeichner nicht initialisiert und wird deswegen `&null` sein. In den meisten Fällen folgt dann ein Laufzeitfehler.

### 8.1.3 Prozedur-Handling

#### *8.1.3.1 Procedure-Aufruf*

Eine Prozedur `procedurename` wird durch Prozeduraufrufe aufgerufen wie:

```
procedurename(expr1, expr2, …, exprn)
procedurename !liste_oder_record
```

#### *8.1.3.2 Von einer Prozedur zurückkehren*

Eine Prozedur kehrt zum Aufrufer zurück, sobald die Zeile

```
return ausdruck
```

angetroffen wird. Wenn es keinen `ausdruck` gibt, dann wird `&null` zurückgegeben. Falls die Auswertung von `expression` scheitert, dann scheitert die Prozedur ebenso.

Wenn Du möchtest, dass eine Prozedur zurückkehrt und scheitert, dann kannst Du entweder `return &fail` oder `fail` anwenden.

Nach Rückkehr mittels `return` oder `fail` existieren alle dynamischen lokalen Bezeichner nicht mehr.

Wenn Du möchtest, dass die dynamischen lokalen Bezeichner bis zum nächsten Aufruf dieser Prozedur intakt sein sollen, dann musst Du zurückkehren durch Anwendung von

```
suspend expression1 do expression2
```

Listing: home/pi/icon9_51/bin/suspend.icn

```
1  procedure suspend_test(n)
2     suspend 1 to n do write(" up")
3     suspend n to 1 by -1 do write(" down")
4  end
5
6  procedure main()
7     every writes(suspend_test(5))
8  end
```

| # | Erläuterung |
|---|---|
| 2 | **Syntax: suspend expression1 [do expression2]**<br>Die Prozedur gibt kein einziges Ergebnis zurück. Es gibt mehrere Werte, die von `expression1` bereitgestellt werden. Der Aufrufer kann nach einem Ergebnis fragen, der Aufrufer kann aber auch nach so vielen Ergebnissen fragen, wie sie möglich sind.<br>Wenn Du weitere Zeilen in der Art `suspend sin(&pi/4.0)` hinzufügst, wirst Du weitere Ergebnisse erhalten. |
| 7 | Dieser Ausdruck erhält so viele Ergebnisse, wie die Prozedur `suspend_test()` erzeugen kann. Falls es in `suspend_test()` keine weiteren Ergebnisse mehr gibt, dann scheitert der Ausdruck in Zeile 7 – das Programm endet. |

`return i+j` ist das Gleiche wie `return (i+j)`

```
return
ausdruck
```

Der Code gibt `&null` zum Aufrufer zurück. `ausdruck` wird niemals ausgewertet.

#### *8.1.3.3 Prozeduren als Werte*

Da Prozeduren ja global deklarierte Werte sind, können sie Bezeichnern zugewiesen werden, als Argumente übergeben werden usw.

home/pi/icon9_51/bin/procedure_as_value.icn

```
1  link graphics
2
3  procedure writes(s)
4      WOpen("size=800,640")
5      Font("URW Chancery L,20")
6      GotoRC(3,2)
7      WWrite(s)
8      GotoRC(5,2)
9      WWrite(Font())
10     GotoRC(10,2)
11     WWrite("Hit any key")
12     Event()
13     WDelay(500)
14     WClose()
15 end
16
17 procedure main()
18     trigon    :=    list(3,"sin","cos","tan")
19     say := write("Hi there!")
20     say
21     write(*say)
22     say := write
23     say("How do you do?")
24     writes("Another message")
25     writes(trigon[2](&pi/4.0))
26 end
```

| # | Erläuterung |
|---|---|
| 3 | Es gibt eine in Icon eingebaute Prozedur des gleichen Namens – sei gewiss, dass wir hier recht sonderbare Dinge treiben. Sowas wie das hier solltest Du allerdings nicht machen. Ich will nur zeigen, was so alles möglich ist.<br>Somit wird jedes Mal, wenn wir `writes()` aufrufen, diese Funktion aufgerufen – und nicht die eingebaute. |
| 4 - 14 | Wir werden alle Funktionen, die mit einem Großbuchstaben beginnen, besprechen, sobald wir GUIs ab Kapitel 15 programmieren. Diese Zeilen öffnen ein Fenster, setzen den Zeichensatz, positionieren den Textcursor, schreiben die Zeichenkette, die als Parameter für diese Prouzedur angegeben wurde, schreiben den Namen des Zeichensatzes, geben eine Meldung ab, was als nächstes zu tun ist, warten auf ein Ereignis, verzögern die Programmausführung für 500 ms und schließen das Fenster am Ende. |
| 18 | Eine Liste, die die Zeichenkettennamen von drei trigonometrischen Funktionen enthält, wird deklariert |
| 19 | Die Prozedur `write()` wird dem Bezeichner `say` zugewiesen. Üblicherweise schreibt die Funktion `write()` irgendetwas und gibt das letzte Argument als Ergebnis zurück. Was wird der Inhalt des Bezeichners `say` sein, vielleicht die Zeichenkette oder die ganze Funktion? |
| 20 | Das einfache `say` macht das Gleiche, wie es die Funktion `write()` machen würde |
| 21 | Das Ergebnis ist 9 – die Länge der in Zeile 19 erwähnten Zeichenkette |
| 22 | Es gibt eine weitere Zuweisung zum Bezeichner `say` – jetzt gibt es nur den Prozedurnamen ohne (). |
| 23 | Das Ergebnis ist das Gleiche, als wenn wir `write()` anwenden würden |
| 24 | Unsere Prozedur, die wir am Anfang definiert haben, wird aufgerufen. Nachdem Du eine Taste auf der Tastatur gedrückt hast, wird das Fenster geschlossen. |
| 25 | Unsere Prozedur wird nochmals aufgerufen. Das Argument wird als die Liste definiert, die wir in Zeile 18 definiert haben. Das zweite Element ist der Name der Funktion Cosinus. Diese Funktion erhält das Argument π/4. Somit gehen wir davon aus, dass das Ergebnis dieser Berechnung in ein neues Fenster geschrieben werden wird. |

## 8.1.4 Variablen und Dereferenzierung

Variablen können sein

- Bezeichner (global, lokal, statisch)
- Elemente von Listen und Tabellen
- Felder von Records
- Bereiche aus Zeichenketten
- Schlüsselworte

Listing: home/pi/icon9_51/bin/dereferencing.icn

```
1  procedure main()
2      line := "init"
3      write("Dereferencing demo")
4      write("Enter something, please:")
5      write(line, " without dereferencing ",
6                line := read())
7
8      line := "init"
9      write(.line, " with dereferencing ",
10               line := read())
11 end
```

| # | Erläuterung |
|---|---|
| 5 | Der Wert des Bezeichners `line` wird geschrieben, zusammen mit einer Zeichenkette und nachdem die Eingabe beendet ist. Der Wert für `line` wird im ersten und letzten Parameter der gleiche sein – wie wir es von früheren Beispielen bereits kennen. |
| 9 | Zeile 9 ist Zeile 5 ähnlich – der einzige Unterschied ist der sogenannte Dereferenzierungs-Operator `.` vor dem Bezeichner `line`. Dies bedeutet, dass der Bezeichner `line` zuerst dereferenziert wird, der aktuelle Wert wird für das erste Argument der Funktion `write()` verwendet – unabhängig davon, was mit diesem Bezeichner während der Auswertung der anderen Argumente geschehen wird. Das zweite Argument ist eine Zeichenkette. Das dritte Argument ist wieder der Bezeichner `line`, der als Ergebnis einer Eingabe zugewiesen wurde. Falls die Eingabe etwas anderes als die Zuweisung in Zeile 8 ist, werden wir hier ein Beispiel haben, in dem der Bezeichner `line` unterschiedliche Werte in einer Programmzeile enthält. Diese Technik heißt Derefenzierung. |

### 8.1.5 Rekursive Aufrufe

Natürlich kannst Du rekursive Funktionen programmieren; das sind Funktionen, die sich selbst aufrufen. Es gibt zwei wohlbekannte Beispiele in der Algorithmenprogrammierung. Das erste ist die Berechnung der Fakultät, bei dem wir schon gezeigt haben, dass es ohne jegliche Rekursionstechnik programmiert werden kann. Das andere Beispiel ist die Berechnung der Fibonacci-Zahlen, die definiert sind als:

F(1) := 1, F(2) := 1

sonst F(n) := F(i – 1) + F(i - 2)

Listing: home/pi/icon9_51/bin/fibonacci.icn

```
1   procedure fib(n)
2      if n = (1 | 2) then return 1
3      else return fib(n - 1) + fib(n - 2)
4   end
5
6   procedure main()
7      start := &time
8      every i := 1 to 30 do
9      {  write(i, "\t", fib(i))
10     }
11     write("Time of execution: ",
12                &time - start, " ms")
    end
```

| # | Erläuterung |
|---|---|
| 7 | `&time` ist das Schlüsselwort für den Timer, der die Millisekunden seit Programmstart zählt |
| 8 - 12 | Wir wollen 30 Fibonacci-Zahlen berechnen. Die ersten Fibonacci-Zahlen werden innerhalb weniger Hundert ms berechnet – aber die weiteren Zahlen verschlingen eine Menge Zeit. Auf meinem Raspberry Pi werden 13 Sekunden benötigt, um die ersten 30 Fibonacci-Zahlen zu berechnen. |

## 8.2 Co-Expression [Fortgeschrittene]

Der gewöhnliche Programmierstil erlaubt nicht ausdrücklich, einen Ausdruck wieder aufzunehmen, um ein Ergebnis zu erhalten. Die von einem Ausdruck erzeugten Ergebnisse sind in der Stelle und in der Folge der Programmauswertung streng gezwungen.

Eine Co-Expression kennt diese Beschränkungen nicht, weil sie einen Ausdruck einnehmen, um zu einem beliebigen Zeitpunkt und Ort ausdrücklich wieder aufgenommen zu werden. Eine Co-Expression ist ein Datenobjekt, das eine Referenz zu einem Ausdruck enthält sowie eine Umgebung, um diesen Ausdruck auszuwerten.

### 8.2.1 Co-Expression-Operationen

Zunächst müssen Co-Expressionen durch die Kontrollstruktur

```
create expression
```

erzeugt werden, die eine Co-Expression erzeugt, die auf `expression` verweist. Diese Co-Expression kann einer Variablen zugewiesen werden, als Argument einer Prozedur übergeben werden oder als Ergebnis einer Prozedur zurückgegeben werden – so wie jeder andere Wert auch. Das oben erwänte Datenobjekt enthält ebenfalls eine Kopie der dynamischen lokalen Variablen der Prozedur, in der `create` erscheint. Dies befreit `expression` aus dem Programmspeicher, wo es erscheint und mit einer Umgebung von sich selbst zur Verfügung stellt.

Beispiel: `decimal:= create(0 to 255)`

### 8.2.2 Aktivieren von Co-Expressions

Die Steuerung wird einer Co-Expression durch ihre Aktivierung übertragen, die durch den Aktivierungsausdruck `@Co` vorgenommen wird, was wiederum verursacht, dass die Programmausführung in dem Ausdruck fortfährt, auf die durch `Co` verwiesen wird. Sobald `Co` ein Ergebnis erzeugt, kehrt die Steuerung zum aktivierenden Ausdruck zurück und das erzeugte Ergebnis wird zum Ergebnis des Akltivierungsausdrucks.

Beispiel: `write(right(@decimal, 4))`

Falls diese Co-Expression zum ersten Mal aktiviert wird, wird es das ASCII-Zeichen mit dem Ordinalwert 0 erzeugen. Beim nächsten Mal, wenn die Co-Expression wieder aktiviert wird,

wird das Ergebnis das ASCII-Zeichen mit dem Ordinalwert 1 sein. … Nach dem Verbrauchen des letzten Wertes (ASCII-Zeichen mit dem Ordinalwert 255) gibt es keine weiteren Ergebnisse – die nächste Aktivierung scheitert.

### 8.2.3 Auffrischen von Co-Expressions

Die Operation `^Co` erzeugt eine Kopie einer Co-Expression `Co`. Beispiel: `Co := ^Co`.

### 8.2.4 Anzahl der erzeugten Werte

Die Anzahl der Aktivierungen einer Co-Expression liefert der Größen-Operator *: `*Co`

### 8.2.5 Verwenden von Co-Expressions

#### *8.2.5.1 Kennzeichnungen und Markierungen*

Stell Dir vor, Du möchtest einmalige Kennzeichnungen (Label) und Markierungen (Tags) erzeugen. Unter Anwendung herkömmlicher Techniken würdest Du etwas nutzen wie

```
procedure unique_label()
   static label
   initial label := 0
   return "Label" || (label +:=1)
end
```

Unter Verwendung von Co-Expressions würdest Du `label := create ("Label" || seq())` haben. Wenn Du ein einmaliges Label brauchst, dann musst Du ein `@label` nehmen.

#### *8.2.5.2 Parallel-Abläufe*

Wir wollen ein Programm haben, das die Tabelle

```
 64    01000000    40    100    "@"    @
 65    01000001    41    101    "A"    A
 66    01000010    42    102    "B"    B
 67    01000011    43    103    "C"    C
 68    01000100    44    104    "D"    D
 69    01000101    45    105    "E"    E
```

darstellt. Ohne Co-Expressions müsstest Du Umwandlungsroutinen anwenden, um die Binär-, Hexadezimal- bzw. Oktaldaten sowie Zeichen zu erhalten, die in den letzten beiden Spalten angezeigt werden. Bei Nutzung von Co-Expressions brauchst Du keine Umwandlungsroutinen – und das Programm wird recht kurz!

Du wirst Co-Expressions für jeden Generator erzeugen müssen, der die Spalten ergibt – und Du musst diese Co-Expressions parallel aktivieren.

Listing: home/pi/icon9_51/bin/coexpression.icn

```
1   procedure main()
2      decimal:= create(0 to 255)
3      bin    := create(!"01" || !"01" ||
4                       !"01" || !"01" ||
5                       !"01" || !"01" ||
6                       !"01" || !"01")
7      hex    := create(!"0123456789ABCDEF" ||
8                       !"0123456789ABCDEF")
9      octal  := create((0 to 3) ||
10                      (0 to 7) || (0 to 7))
11     escape := create image(!&cset)
12     cha    := create (!&cset)
13
14     while write(right(@decimal, 4), "\t",
15                 right(@bin, 10), "\t",
16                 right(@hex, 4), "\t",
17                 right(@octal, 5), "\t",
18                 right(@escape, 10), "\t",
19                 right(@cha, 4))
20  end
```

| # | Erläuterung |
|---|---|
| 2-12 | Co-Expressions für jeden Generator, um die Elemente jeder Spalte zu liefern |
| 14-19 | Die Co-Expressions werden parallel aktiviert |

Ein anderes Beispiel, das Co-Expressions in Parallel-Programmierung nutzt: Erinnere Dich an die Liste, die wir genutzt haben, um Argumente zuzuweisen, die durch einen Programmaufruf übergeben wurden. Dies ist ebenfalls durch Co-Expressions möglich.

Listing: home/pi/icon9_51/bin/argumenter.icn

```
1   procedure main(args)
2      argumenter := create !args
3
4      every (a | b | c | d) := @argumenter
5      write(name(a),": ", a)
6      write(name(b),": ", b)
7      write(name(c),": ", c)
8      write(name(d),": ", d)
9      # Add some code to set defaults
10     # if not set included in args
11  end
```

| # | Erläuterung |
|---|---|
| 1 | `args` ist eine Liste, die alle Argumente (Parameter) enthält |
| 2 | Eine Co-Expression `argumenter` wird erzeugt |
| 4 | Die Co-Expression wird aktiviert |
| 5 | **Syntax: name(v):s erzeugt einen Namen**<br>`name(v)` erzeugt den Namen der Variablen `v` |

### 8.2.6 Programmierer-definierte Kontrollstrukturen

Co-Expressions ermöglichen es, Icon's eingebaute Kontrollstrukturen zu erweitern. Lasst uns ein paar Zeilen Code eingeben.

Listing: home/pi/icon9_51/bin/coexp_PDCS.icn

```
1  procedure parallel(C1, C2)
2      local x
3
4      repeat
5      {   if x := @C1 then suspend x else fail
6          if x := @C2 then suspend x else fail
7      }
8  end
9
10 procedure Parallel(L)
11     local x
12
13     repeat
14     {   if x := @L[1] then suspend x
15         else fail
16         if x := @L[2] then suspend x
17         else fail
18     }
19 end
20
21 procedure main()
22     every writes(parallel(create !&lcase,
23                           create !&ucase))
24     write()
25     every writes(Parallel([create !&lcase,
26                           create !&ucase]))
27     write()
28     every writes(Parallel{!&lcase, !&ucase})
29     write()
30     every writes(Parallel{!&lcase, !&ucase})
31 end
```

| # | Erläuterung |
|---|---|
| 1 | C1 und C2 sind Co-Expressions, die in einem Prozedur-Aufruf erzeugt werden |
| 10 | L ist eine Lliste von Co-Expressions, die in einem Prozedur-Aufruf erzeugt werden |
| 22<br>23 | Co-Expressions werden innerhalb eines Prozeduraufrufs erzeugt (*Anrufung*) |
| 25<br>26 | Co-Expressions werden innerhalb einer Liste, die einer Prozedur übergeben werden, erzeugt und aktiviert |
| 28 | Eine weitere Syntax, Co-Exxpressions zu erzeugen und zu aktivieren |
| 30 | Durch Nutzung dieser Technik wird Auffrischen nicht verlangt, um die Co-Expression wieder zu starten |

### 8.2.7 Andere Merkmale

#### *8.2.7.1 Übetragung der Programmsteuerung zwischen Co-Expressions*

Wir wollen einige Co-Expressions erzeugen, die die Programmsteuerung zwischen diesen übertragen. Und wir wollen, dass die Verbindungen von einer Co-Expression zu einer anderen jedes Mal geändert wird, wenn diese Co-Expression die Programmsteuerung erhält. Lasst uns wieder ein paar Zeilen Code eingeben – wir werden erkennen, dass die dynamischen lokalen Bezeichner die gleichen sind wie beim Verlassen dieser Co-Expression.

Listing: home/pi/icon9_51/bin/coexp_transfer.icn

```
1  global Co
2
3  procedure p(C, t)
4      local i
5
6      i := 0
7      repeat
8      {   write(repl(" ", i),"Activation ",
9                i +:= 1, " of ", t)
10         C :=: Co[?10]
11         delay(100)
12         @C
13     }
14 end
15
16 procedure main()
17     Co := list(10)
18     index := set([1,2,3,4,5,6,7,8,9,10])
19     every i := 1 to 10 do
20     {   c := ?index
21         Co[i] := create p(Co[c], "Co-Exp "
22                                || string(c))
23         write(i,"\t", image(Co[i]),"\t", c)
24         index --:= set([c])
25     }
26     read()
27     @Co[1]
28 end
```

| # | Erläuterung |
|---|---|
| 17 | Eine Liste wird definiert, 10 Co-Expressions zu enthalten |
| 18 | Eine Menge wird definiert, die die ersten 10 Zahlen enthält |
| 21<br>22 | Eine Co-Expression wird erzeugt, die aus einer Prozedur mit zwei Parametern besteht, einer Co-Expression und einer Zeichenkette |
| 24 | Die Menge wird um das zufällig ausgewählte Element verkleinert |
| 27 | Die mit 1 indizierte Co-Expression wird aktiviert, daher erhält die Funktion `p`, die in der Erzeugung verwendet wurde, die Programmsteuerung |
| 3 | Prozedur `p` besitzt zwei Argumente, eine Co-Expression und eine Zeichenkette - genau, was wir in der Erzeugung festgelegt hatten (Zeilen 21, 22) |
| 4 | Eine lokale Variable wird definiert – diese hier ist die dynamische lokale Variable, die als Umgebung für das nächste Mal, wenn `p` mit der gleichen Co-Expression aktiviert wird, verwendet wird |
| 10 | Die Co-Expression, die als Argument zu `p` angegeben ist, wird einer anderen zufällig ausgewählten Co-Expression zugewiesen ... |
| 12 | ... welche aktiviert wird |

### *8.2.7.2 Eingebaute Co-Expressions*

Es gibt drei eingebaute Co-Expressions, die als Schlüsselworte definiert sind

- `&source`: Co-Expression, die die aktuell aktive Co-Expression aktiviert hat. `@&source` kehrt zur aktivierenden Co-Expression zurück.
- `&current`: Co-Expression, in der die Programmausführung stattfindet.
- `&main`: Co-Expression für den Aufruf der Prozedur `main()`. `@&main` gibt die Steuerung der Co-Expression für die Prozedur `main()` an dem Punkt zurück, an dem es eine Co-Expression aktivierte. Das kann sonstwo im Programm sein, z.B. In einer Prozedur, die die Co-Expression aktivierte.

Listing: home/pi/icon9_51/bin/coexp_builtin.icn

```
1  global Co
2
3  procedure p(C, t)
4     local i
5
6     i := 0
7     repeat
8     {  write(repl(" ", i), image(&source),
9              " --> ", image(&current),
10             " Activation #", i +:= 1)
11       if i = 10 then @&main
12       C :=: Co[?10]
13       delay(100)
14       @C
15    }
16 end
17
18 procedure main()
19    Co := list(10)
20    index := set([1,2,3,4,5,6,7,8,9,10])
21    every i := 1 to 10 do
22    {  c := ?index
23       Co[i] := create p(Co[c],
24                         "Co-Exp " || string(c))
25       write(i,"\t", image(Co[i]),"\t", c)
26       index --:= set([c])
27    }
28    read()
29    @Co[1]
30    write("Co-expression ", image(&source),
31             " was activated 10 times")
32    write("Control was given to ",
33          image(&current),
34          "and is now in procedure ",
35          image(&main), " again")
36 end
```

| # | Erläuterung |
|---|---|
| 8, 9 | Anwendung der Schlüsselworte `&source` und `&current` |
| 11 | Anwendung von `&main`, `@&main` bedeutet Rückkehr zur ersten Aktivierung einer Co-Expression, die von der Prozedur `main()` kommt – Ausführungskontrolle kehrt zur Zeile hinter Zeile 29 zurück |
| 30 | Anwendung von `&source` |
| 33 | Anwendung von `&current` |
| 35 | Anwendung von `&main` |

### *8.2.7.3 Übermittlung*

Wir haben Co-Expressions genutzt, um Prozeduren aufzurufen, die wir für die Erzeugung der Co-Expression nahmen. Übermittlung bedeutet, dass wir Co-Expressions zusammen mit Icon's Ausdrücken nutzen können. Wir nehmen ein Beispiel aus dem Icon-Buch, das vorstellt, wie Co-Routine-Programmierung zu nutzen ist.

Listing: home/pi/icon9_51/bin/coexp_coroutines.icn

```
1  global words, lines, writer
2
3  procedure main()
4     words     := create word()
5     lines     := create reader()
6     writer    := create output()
7     @writer
8  end
9
10 procedure word()
11    while line := @lines do
12       line ? while tab(upto(&letters)) do
13                    tab(many(&letters))
14    @writer
15 end
16
17 procedure reader()
18    while read() @words
19 end
20
21 procedure output()
22    while write(@words)
23    @&main
24 end
```

## 8.2.8 Bibliotheken

Falls Du in die Icon-Quellen schauen möchtest, die Programmierer-definierte Steuerungsoperationen verwenden, kannst Du nehmen oder einbinden

- `pdco` für Prozeduren für verschiedene Steuerungsoperationen
- `pdae` für Prozeduren für Programmiererdefinierte Argument-Auswertungsmethoden

**Übungsaufgabe:**

Nimm das Programm, das wir für die Berechnung der Fibonacci-Zahen verwendet haben und überlege Dir, wie Du es mit den Kenntnissen, die Du heute und beim letzten Mal erworben hast, beschleunigen könntest. Mit beschleunigen meine ich jetzt nicht, das Programm um 5 oder 20 % – sondern um Größenordnungen schneller zu machen!

Anstatt jede Zahl wieder und immer wieder zu berechnen, speichere sie in eine Tabelle. Nimm den der Tabelle zugewiesenen Wert, sofern vorhanden – und berechne sie, falls sie noch nicht existiert.

Du könntest eine `static`-Deklarierung, eine `local`-Deklarierung und eine `initial`-Klausel gebrauchen und Du musst die Prozedur `fib()` ändern, damit sie mit Werten umgehen kann, die Tabellen zugewiesen sind.

Übrigens: Du wirst sehr erstaunt sein, wie schnell Dein Algorithmus arbeiten kann! Versuche, die Anzahl der Schleifen zu erhöhen. Du hast jetzt ein weiteres Beispiel für Large Integer Arithmetik (LIA).

Das war's für Heute! Genießt Euer Leben bis zum nächsten Mal!

Was kommt demnächst?

9 Konzept des Zeichenketten-Scans