- Gut gemeint – aber leider fehlerhaft ...
- Jeder Producer benötigt zwei Locks gleichzeitig, um zu produzieren:
  1. dasjenige für den Puffer;
  2. dasjenige für einen Semaphor.
- Muss er für den Semaphor ein `wait()` ausführen, gibt er das Lock für den Semaphor wieder zurück ... nicht aber dasjenige für den Puffer !!!
- Die Folge ist, dass niemand mehr eine Puffer-Operation ausführen kann, insbesondere auch kein `up()` mehr für den Semaphor $\Longrightarrow$ Deadlock

Anwendung (2. Versuch :-) Entkopplung der Locks

```
    ...
    public void produce(Data d) throws InterruptedException {
       free.down();
       synchronized (this) {
          a[last] = d; last = (last+1)%cap;
       }
       occupied.up();
    }
    public Data consume() throws InterruptedException {
       Data result; occupied.down();
       synchronized (this) {
          result = a[first]; first = (first+1)%cap;
       }
       free.up(); return result;
    }
}   // end of corrected class Buffer
```

- Das Statement `synchronized ( obj ) { stmts }` definiert einen kritischen Bereich für das Objekt obj, in dem die Statement-Folge stmts ausgeführt werden soll.
- Threads, die die neuen Objekt-Methoden `void produce(Data d)` bzw. `Data consume()` ausführen, benötigen zu jedem Zeitpunkt nur genau ein Lock :-)

Warnung:

Threads sind nützlich, sollten aber nur mit Vorsicht eingesetzt werden. Es ist besser,

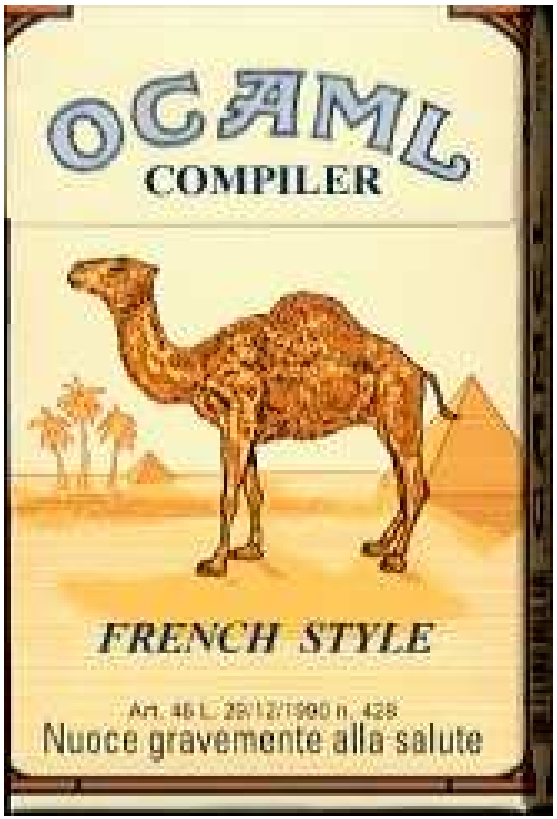
- ... wenige Threads zu erzeugen als mehr.
- ... unabhängige Threads zu erzeugen als sich wechselseitig beeinflussende.
- ... kritische Abschnitte zu schützen, als nicht synchronisierte Operationen zu erlauben.
- ... kritsche Abschnitte zu entkoppeln, als sie zu schachteln.

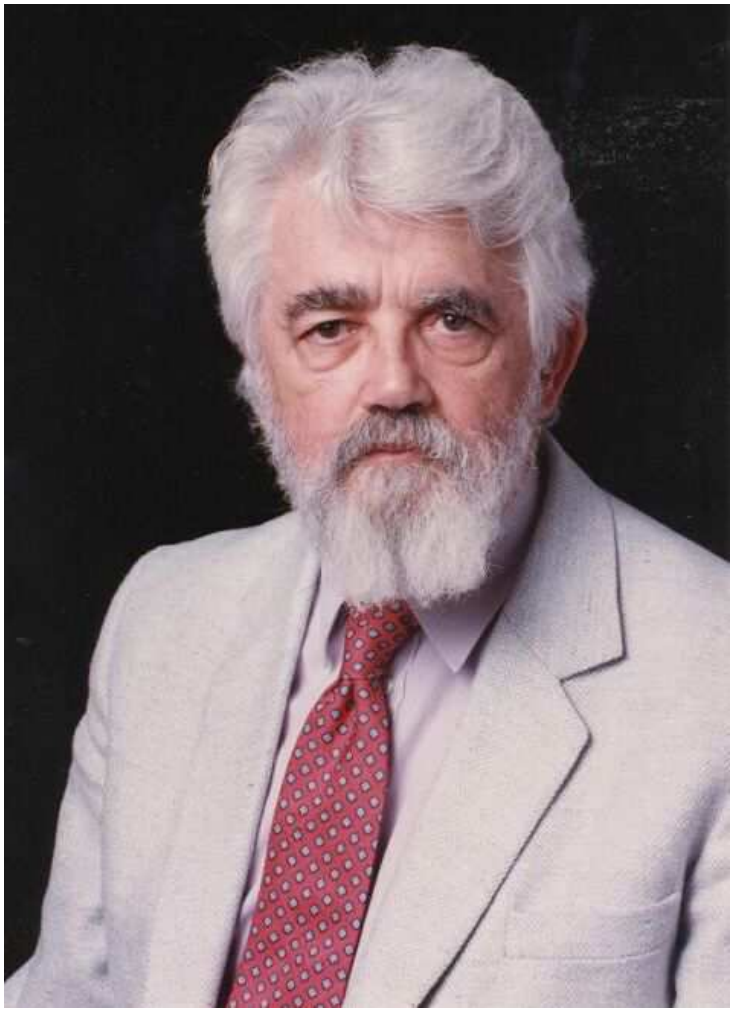
Warnung (Forts.):

Finden der Fehler bzw. Überprüfung der Korrektheit ist ungleich schwieriger als für sequentielle Programme:

- Fehlerhaftes Verhalten tritt eventuell nur gelegentlich auf...
- bzw. nur für bestimmte Scheduler :-(
- Die Anzahl möglicher Programm-Ausführungsfolgen mit potentiell unterschiedlichem Verhalten ist gigantisch.

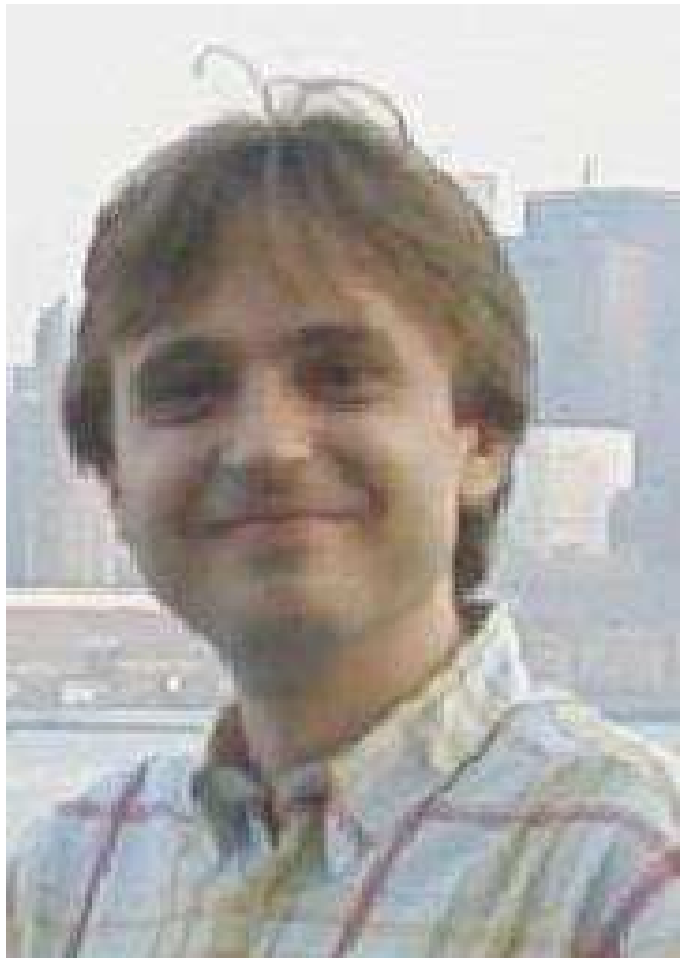
Funktionale Programmierung





John McCarthy, Stanford

Robin Milner, Edinburgh



Xavier Leroy, INRIA, Paris

2 Grundlagen

- Interpreter-Umgebung
- Ausdrücke
- Wert-Definitionen
- Komplexere Datenstrukturen
- Listen
- Definitionen (Forts.)
- Benutzer-definierte Datentypen

2.1 Die Interpreter-Umgebung

Der Interpreter wird mit `ocaml` aufgerufen...

```
seidl@linux:~> ocaml
        Objective Caml version 3.11.1
#
```

Definitionen von Variablen, Funktionen, ... können direkt eingegeben werden :-)

Alternativ kann man sie aus einer Datei einlesen:

```
# #use "Hallo.ml";;
```

2.2 Ausdrücke

```
# 3+4;;
- : int = 7
# 3+
  4;;
- : int = 7
#
```

→ Bei `#` wartet der Interpreter auf Eingabe.

→ Das `;;` bewirkt Auswertung der bisherigen Eingabe.

→ Das Ergebnis wird berechnet und mit seinem Typ ausgegeben.

Vorteil: Das Testen von einzelnen Funktionen kann stattfinden, ohne jedesmal neu zu übersetzen :-)

Vordefinierte Konstanten und Operatoren:

Typ	Konstanten: Beispiele	Operatoren
int	`0 3 -7`	`+ - * / mod`
float	`-3.0 7.0`	`+. -. *. /.`
bool	`true false`	`not || &&`
string	`"hallo"`	`^`
char	`'a' 'b'`	

Typ	Vergleichsoperatoren
int	= <> < <= >= >
float	= <> < <= >= >
bool	= <> < <= >= >
string	= <> < <= >= >
char	= <> < <= >= >

Typ	Vergleichsoperatoren
int	= <> < <= >= >
float	= <> < <= >= >
bool	= <> < <= >= >
string	= <> < <= >= >
char	= <> < <= >= >

```
# -3.0/.4.0;;
- : float = -0.75
# "So"^" "^"geht"^" "^"das";;
- : string = "So geht das"
# 1>2 || not (2.0<1.0);;
- : bool = true
```

2.3 Wert-Definitionen

Mit `let` kann man eine Variable mit einem Wert belegen.

Die Variable behält diesen Wert für immer :-)

```
# let seven = 3+4;;
val seven : int = 7
# seven;;
- : int = 7
```

Achtung: Variablen-Namen werden klein geschrieben !!!

Eine erneute Definition für `seven` weist nicht `seven` einen neuen Wert zu, sondern erzeugt eine neue Variable mit Namen `seven`.

```
# let seven = 42;;
val seven : int = 42
# seven;;
- : int = 42
# let seven = "seven";;
val seven : string = "seven"
```

Die alte Definition wurde unsichtbar (ist aber trotzdem noch vorhanden :-)

Offenbar kann die neue Variable auch einen anderen Typ haben :-)

2.4 Komplexere Datenstrukturen

- Paare:

```
# (3,4);;
- : int * int = (3, 4)
# (1=2,"hallo");;
- : bool * string = (false, "hallo")
```

- Tupel:

```
# (2,3,4,5);;
- : int * int * int * int = (2, 3, 4, 5)
# ("hallo",true,3.14159);;
-: string * bool * float = ("hallo", true, 3.14159)
```

Simultane Definition von Variablen:

```
# let (x,y) = (3,4.0);;
val x : int = 3
val y : float = 4.

# let (3,y) = (3,4.0);;
val y : float = 4.0
```

Records: Beispiel:

```
# type person = {vor:string; nach:string; alter:int};;
type person = { vor : string; nach : string; alter : int; }
# let paul = { vor="Paul"; nach="Meier"; alter=24 };;
val paul : person = {vor = "Paul"; nach = "Meier"; alter = 24}
# let hans = { nach="kohl"; alter=23; vor="hans"};;
val hans : person = {vor = "hans"; nach = "kohl"; alter = 23}
# let hansi = {alter=23; nach="kohl"; vor="hans"}
val hansi : person = {vor = "hans"; nach = "kohl"; alter = 23}
# hans=hansi;;
- : bool = true
```

Bemerkung:

... Records sind Tupel mit benannten Komponenten, deren Reihenfolge irrelevant ist :-)

... Als neuer Typ muss ein Record vor seiner Benutzung mit einer `type`-Deklaration eingeführt werden.

... Typ-Namen und Record-Komponenten werden klein geschrieben :-)

Bemerkung:

... Records sind Tupel mit benannten Komponenten, deren Reihenfolge irrelevant ist :-)

... Als neuer Typ muss ein Record vor seiner Benutzung mit einer `type`-Deklaration eingeführt werden.

... Typ-Namen und Record-Komponenten werden klein geschrieben :-)

Zugriff auf Record-Komponenten

... per Komponenten-Selektion:

```
# paul.vor;;
- : string =  "Paul"
```

... mit Pattern Matching:

```
# let {vor=x;nach=y;alter=z} = paul;;
val x : string = "Paul"
val y : string = "Meier"
val z : int = 24
```

... und wenn einen nicht alles interessiert:

```
# let {vor=x} = paul;;
val x : string = "Paul"
```

Fallunterscheidung: `match` und `if`

```
match n
 with 0 -> "Null"
    | 1 -> "Eins"
    | _ -> "Soweit kann ich nicht zaehlen!"

match e
 with true  -> e1
    | false -> e2
```

Das zweite Beispiel kann auch so geschrieben werden (-:

```
if e then e1 else e2
```

Vorsicht bei redundanten und unvollständigen Matches!

```
# let n = 7;;
val n : int = 7
# match n with 0 -> "null";;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
Exception: Match_failure ("", 5, -13).
# match n
   with 0 -> "null"
      | 0 -> "eins"
      | _ -> "Soweit kann ich nicht zaehlen!";;
Warning: this match case is unused.
- : string = "Soweit kann ich nicht zaehlen!"
```