Programmiersprachen

Vorlesung 4

Zusammenfassung

Georg Ernst Moser

Structuring the Computation

Expressions

- **infix notation :** Operator steht zwischen 2 Operanden. Er kann folglich nur für binäre Operationen eingesetzt werden. Einte triäre Operation wie (x : y ? z) setzt sich in Wirklichkeit aus 2 gleichzeitig angewendeten binären zusammen.
  Bsp: a ∗ (b + c)

- **prefix notation:** Operator steht vor den Operanden.
  Bsp: ∗ a + bc

- **postfix notation :** Operator steht nach den Operanden. Lässt sich einfach mit einem Stack abarbeiten.
  Bsp: abc + ∗

Operatorsymbole können durch Überladen für 2 mögliche Notationen definiert werden. Für alle 3 ist dies nicht möglich, da in diesem Fall die Auflösung eines Ausdruckes nicht mehr möglich wäre.

Beim Auflösen von Infix Operationen werden 2 Merkmale der Operatoren benötigt:

- **Preceedence** (Gewichtung): Welcher Operator bindet stärker als andere (z.b.: Punkt vor Strich)
- **Assoziativität**: Haben 2 Operatoren dieselbe Gewichtung, welcher wird bevorzugt (rechts oder links)

Beispiele:

```
a + b ∗ c   corresponds to    a + (b ∗ c)            (Pascal, C,...  )
```

Fast überall wird „Punkt-“ vor „Strichrechung“ aufgelöst. Dies trifft aber nicht immer zu: In Smalltalk zum Beispiel wird immer von links nach rechts ausgewertet (Klammern setzen).

```
a = b < c   corresponds to    (a = b) < c            (Pascal)
a == b < c  corresponds to    a == (b < c)           (C)
```

Der Kleiner Vergleich ist in Pascal und C unterschiedlich gewichtet.

conditional expressions

```
(a > b) ? a : b                                      (C)
if a > b then a else b                               (ML)
case x of 1 => f1(y) | 2 => f2(y) | => g(y)          (ML)
```

Conditional Statements

If statements

Dangling else Problem: Steht ein *else* nach 2 geschachtelten *if* statements, zu welchem gehört das *else*?
Tritt in vielen Sprachen auf, vor allem aber in Verbindung mit C Syntax.
Mit einem *begin* und *end* statement kann dieses Problem beseitigt werden, da die Auflösung hierdurch eindeutig wird.

Beispiele:

```
If x > 0 then if x < 10 then x := 0 else x := 1000
//das end statement schließt den then "block"
If x > 0 then begin if x < 10 then x := 0 end else x := 1000
If x > 0 then if x < 10 then x := 0 end else x := 1000 end
```

Eine Verschachtlung von vielen *if* und *else* Statements durch Einrücken ist aus Platz- und Übersichtsgründen nicht möglich.

```
if(x=1){
      else{
            if(x=2){
                  else{
                        //weitere Einrückungen...
                  }
            }
      }
}
```

Aus diesem Grund wurde das *else if* eingeführt. Dies ist jedoch etwas anderes als ein *if* mit einer folgenden *else* Strucktur. In diesem Fall wird zum Abschluss nur ein *end* benötigt.

```
If a then S1 else if b then S2 else if c then S3 else S4 end
```

Switch Statements

```
switch (operator) {
case ’+’: result = operand1 + operand2; break;
case ’-’: result = operand1 - operand2; break;
default: break;
}
```

Am Ende jedes *case* muss ein *break* gesetzt werden, sonnst „fällt“ man in den nächsten Fall durch. Ursprünglich war man der Annahme, der Verzicht auf ein implizites *break* wäre vorteilhaft, da das „Durchrutschen“ in den nächsten Fall vom Programmierer ja bewusst eingesetzt werden könnte. Das Problem der Vergessenen break Anweisung zählt zu den häufigsten Programmierfehlern überhaupt.
Praktisch alle neueren Sprachen, welche über eine eingeständige Syntax verfügen, verzichten deswegen auf ein explizites *break*.

Gleiches Beispiel in **Ada**:

```
case OPERATOR is
      when ’+’ => result := operand1 + operand2;
      when ’−’ => result := operand1 − operand2;
      when others => null;
end case
```

Loops - Schleifen

For - Schleife

For Schleifen sind in fast jeder Sprache vorhanden, obwohl diese rein semantisch nicht benötigt werden. Sie könnten immer zum Beispiel durch eine *while* Schleife ersetzt werden.

Beispiele:

```
for var := lower to upper do statement                    (Pascal)
```

Ursprüngliche Form einer for Schleife: Eine Laufvariable wird deklariert und Ober- und Untergrenze angegeben (eventuell auch ein increment in Form von *to upper by x*)

```
for var in discrete range loop body end loop              (Ada)
```

Auch in Ada ist dies syntaktisch sehr ähnlich. Der *range* ist eine Menge. Statt *do* wird hier *loop body* geschrieben. Es wird über die Menge (*range*) iteriert, ähnlich einer *for-each* Schleife. Eine Schleife in Form von *loop* kann in Ada auch ohne Präfix (wie *for* …) stehen. Dies ist dann eine Endlosschleife (kein *while(true)*,... nötig).

```
for (int i = 0; i < 10; i++) {. . . }                    (C++)
```

Die C Syntax weicht deutlich von der der ursprünglichen for Schleife ab. Dies kommt daher, dass in C eigentlich keine *for* Schleife geplant war („wir haben sowieso schon alles was wir brauchen, wofür?“). Diese wurde nachträglich hinzugefügt, als rein syntaktische Vereinfachung.

```
for (
int i = 0;        //statement welches vor Block geschrieben wird
i < 10;           //while Schleife
i++               //am Ende des Blocks
) {. . . }
```

while – Schleife

Die wichtigste Form der Schleife, da sie in jedem Fall eingesetzt werden kann. Sie ist auch in jeder Sprache mit Schleifen enthalten.
Die Syntaxunterschiede zwischen den einzelnen Sprachen sind minimal:

```
while condition do statement                             (Pascal)
while (expression) statement;                            (C)
while condition loop loop body end loop                  (Ada)
```

Beim *repeat* Statement wird die Bedingung am Ende überprüft. Diese Form der Schleife ist streng genommen nicht notwendig, weshalb viele Sprachen auch keine Unterstützung dafür haben. Auch die Syntax unterscheidet sich von Sprache zu Sprache stark:

```
repeat statement until condition                         (Pascal)
do statement while (expression);                         (C)
loop statement; exit when condition end loop             (Ada)
```

Fazit: Sprachkonstrukte die wichtig sind, verfügen in allen Sprachen über eine ähnliche Syntax. Zusätzliche Funktionen sehen überall anders aus.

Es ist auch fast immer möglich aus mehreren Schleifen mit einem Statement auszusteigen (Beispiel in Ada) Die Schleife wird benannt (auch in Java). Einige Sprachen verzichten auf dieses Feature, da es die Prinzipien der strukturierten Programmierung verletzt (ähnlich dem *goto*).

```
A: loop . . . loop . . . exit A; . . . end loop . . . end loop A;
```

Routinen

Routinen können in den meisten Sprachen andere Routinen als Eingangsparameter haben.

Beispiel in Pascal

```
//das var vor einem Parameter markiert diesen als
//call-by-reference, sonnst call-by-value
procedure p(var x: T; y: Q; function f(z: R): integer);
```

In vielen davon (auch in Pascal) ist diese Form der Routinenübergabe jedoch abgeschwächt. Routinen sind hier keine first class entities, da es nicht möglich ist Routinen als Rückgabelparameter zu verwenden. Der Grund dafür ist praktischer Natur: Wenn Routinen nur als Eingangsparameter übergeben werden können (nach „innen“), kann es zu keinen Problemen mit dem Stack kommen. Bei der Rückgabe von Routinen (nach „außen“), könnte anschließend auf einen Stack zugegriffen werden, welcher bereits abgebaut wurde.

In Funktionalen Sprachen sind Funktionen meist first class entities. Diese liegen hier wie das gesamte Environment auf dem Heap, weshalb die Übergabe von Funktionen in keinem Fall Probleme bereiten kann.

Beispiel in C Syntax:
In C kann die Übergabe von Funktionen mit Zeigern realisiert werden.

```
void proc(int* x, int y)
{ *x = *x + y; }
```

In C++ gibt es auch sogenannte Referenzen(können nicht null sein). Hier muss nicht dereferenziert werden.

```
void proc(int& x, int y)
{ x = x + y; }
```

Alias Probleme

```
u := x + z + f(x,y) + f(x,y) + x + z
```

f(x,y) kann Seiteneffekte haben → 2x f(x,y) != f(x,y)+f(x,y)
x und y könnten innerhalb von f(x,y) verändert werden. Auch die Auswertungsreihenfolge ist relevant (in C zum Beispiel nicht definiert).

Exception Handling

Exception Handling in Ada

In Ada wurde bereits früh eine Form des Exception-Handlings eingeführt. Es gibt einige wenige vordefinierte Exceptions, welche im Wesentlichen jedoch nur Namen sind:
Constraint Error : Wenn zum Beispiel ein Range für einen Wert definiert wurde und dieser verletzt wird.
Program Error : Tritt auf beim Versuch eine Kontrollstruktur von Ada zu verletzen (Aussteigen aus einer Funktion ohne return statement).
Storage Error : Speicher aus
Tasking Error: Fehler im Bereich der Nebenläufigkeit

Weiters gibt es auch die Möglichkeit selbst Exceptions zu definieren.
Dies geschieht gleich wie bei gewöhnlichen Variablen:

```
Help: exception;
```

Exceptions können dann explizit „geraist“ werden:

```
raise Help;
```

Exception Handling: Jeder Block (beginnt mit *begin* und endet mit *end*) kann über einen Exception Handler verfügen.

```
begin – – block with exception handling
. . . statements . . .
      exception when Help => . . . statements . . .
            when Constraint Error => . . . statements . . .
            when others => . . . statements . . .
end;
```

Wenn in *statements* eine Exception auftritt, kann diese im Exception Teil mit einem case statement behandelt werden. Wird keine entsprechender Handler gefunden wird die Exception weiterpropagiert.

In Ada gibt es keine Exception Hierarchien.

Implementation of Exception Handling

Der Compiler weist jeder möglichen Exception einen eindeutigen Namen zu.
Die (wie im vorherigen Ada Beispiel) vergeben Namen reichen hierfür nicht aus: Wird innerhalb eines Blocks eine Exception definiert und diese nach außen weitergereicht, hat sie dort keinen Namen mehr. Aus diesem Grund werden globale Namen für Exceptions benötigt.

Damit eine Exception behandelt werden kann muss ein geeigneter Exception Handler dafür gefunden werden. Diese werden von innen (Block in dem wir uns gerade befinden) nach außen (außengelegener Block..) gesucht, und durch den eigentlichen Methodenblock beschränkt. Wird auf eine Methodengrenze gestoßen wird anhand der Aufrufreihenfolge (Dynamic Link) die Exception an die aufrufende Methode übergeben (propagieren von Exceptions).

Auffinden des Zuständigen Handlers:

Grundsätzlich gibt es hierbei 2 verschiedene Möglichkeiten:

1. Der Actication Record wird um ein Feld erweitert, welches auf eine statische Exception Handling Tabelle verweist. Diese enthält Einträge der Form: Exception → entsprechender Handling Code. Wenn in dieser Tabelle kein passender Handler gefunden wird, wird der Activation Record der Methode vom Stack gelöscht und in der aufrufenden Methode weitergesucht.
   Bei jedem Aufruf muss bei diesem Verfahren jedoch der zusätzliche Zeiger geschrieben werden.

2. Im Activation Record wird hier keinerlei zusätzliche Information geschrieben.
   Es wird eine Liste aller Code-Bereiche erstellt, in denen eine Exception auftreten kann: Codeabschnitt geht von x nach y und hat folgende Exception Handling Tabelle. Wird diese Liste sortiert kann mit dem Instruction Pointer und binärer Suche schnell der zuständige Handler gefunden werden. Es treten nur (geringe) Kosten auf, wenn die Exception wirklich geworfen wird.
   Diese Form der Implementierung ist mittlerweile weiter verbreitet, da Exceptions, gemäß ihres Namens nur in Ausnahmefällen verwendet werden sollen (dann dürfen sie Kosten verursachen).

Wie teuer ist Exception Handling?

Dies wurde in Java sehr genau analysiert. Das treueste war hierbei das Erstellen des Objektes, welches die Exception repräsentiert. Durch Optimierungen, welche auf dieses Erstellen verzichten kann die Performance weiter verbessert werden.
Fazit: Exceptions sind billig.

Exception Handling in C++

In C++ können alle Datentypen als Exception verwendet werden (int,array, Instanz einer Klasse,..).

Eine Exception kann mit *throw* geworfen werden:
`throw Help(MSG1);`

Es kann auch angegeben werden welche Exceptions von einer Methode nach außen propagiert werden dürfen.
`void foo() throw(Help, Zerodivide);`
Mittlerweile ist diese Technik jedoch nicht mehr gerne gesehen.
Beim Auftreten einer Exception die nicht weiter propagiert werden kann (nicht in der *throw* Klausel angegeben) wird die globale Routine *unexpected*(muss vom Programmierer implementiert werden) aufgerufen, welche meistens in einem Programmabbruch endet. Wenn innerhalb des gesamten Programms kein handling (kein Handler) für die Exception existiert wird *terminate*(ebenfalls selbst zu implementieren) aufgerufen.
Vor allem aber die Routine *unexpected* ist nicht wirklich sinnvoll: Wenn eine Exception auftritt, deren Auftreten und so auch Behandlung nicht berücksichtigt wurde, so ist der Fehler bereits (auch schon beim Programmieren) passiert. Er kann in diesem Fall nicht wirklich sinnvoll behandelt werden.
Aus diesem Grund sollte die *throw* Klausel nicht mehr verwendet werden.

Example in C++

```
class Help { ... };
class Zerodivide { ... };
      ...
      try { ... }
      catch(Help& msg) {
            switch(msg.kind) {
                  case MSG1:
                        ...;
                  case MSG2:
                        ...;
                  ...
            }
      }
      catch(Zerodivide& z) {
      ...
      }
```

Checked Exceptions

In Java werden sogenannte checked Exceptions verwendet. Der Compiler überprüft, dass keine Exceptions, welche nicht im Kopf einer Methode deklariert wurden, weitergereicht werden.

Dies ist eines der umstrittensten Features von Java: Existiert eine Exception, welche in vielen Bereichen des Programms auftreten kann muss diese fast in jeden Methodenkopf geschrieben werden.
Die Code Wiederverwendung wird auf diese Weise eingeschränkt.

RuntimeException und Error sind in diesem Fall Ausnahmen, da sie auch propagiert werden, ohne vom Programmiere explizit angegeben werden zu müssen.

Finnaly Block

Wurde erstmals in Modulo 3 verwendet und dient vor allem zum „Aufräumen“ nach einem Fehler

```
try { ... }
catch(Exception1 ex) { ... }
catch(Exception2 ex) { ... }
...
finally { ... } // exception in finally supresses that in try
```

Tritt im *finally* Block eine Exception auf, wird diese weitergereicht und die eventuell zuvor aufgetretenen Exceptions (Exception1, Exception2) gehen verloren.
In aktuelleren Versionen kann nachträglich auf aufgetretenen Exceptions, welche nicht propagiert wurden zugegriffen werden.

Ein anderer Lösungsansatz sind try statemets mit Argument (seit Java 7):

```
try (Reader r = new FileReader(path)) { ... }
// r closed; exception in try suppresses that from closing
```

Die Argumente des try Blocks werden an dessen Ende automatische geschlossen. Auf diese Weise kann auf den *finally* Block verzichtet werden.
Als Argument können Klassen verwendet werden, welche das Interface *Autocloseable* implementieren. Tritt in *close* eine Exception auf geht diese verloren und ursprüngliche Exception wird weitergegeben (genau anders als bei der Verwendung von *finally*).

Exception Handling in ML

Auch in funktionalen Sprachen wird Exception Handling verwendet.

```
exception Neg

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

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

Falls eine Exception auftritt soll das Ergebnis 0 sein.

Exception Handling in Eiffel

Eiffel verwendet die sogenannte retry Semantic. Die einzige Möglichkeit eine Exception zu behandeln ist hierbei, den Codeteil in dem die Exception aufgetreten ist noch einmal auszuführen. In Eiffel gibt es weiters nur eine Art von Exception, welche aus einer fehlgeschlagenen Zusicherung resultiert.

```
try_several_methods is
      local
            i: INTEGER        -- automatically initialized to 0
      do
            try_method(i);
      rescue                  -- executed only when exception occurs
            i := i + 1;
            if i < max_trials then
                  retry       -- retry the execution of try_several_methods
            end
      end
```

Durch die Erhöhung von *i* kann in *try_method* ein anderer Codeteil ausgeführt werden. Inhaltlich kann mit diesem System folglich das gleiche realisiert werden wie zum Besipiel in Java,C++... Wird in einem *rescue* block kein *retry* aufgerufen wird die Exception weiter propagiert.

Pattern Matching

Pattern Matching wird vor allem in funktionalen Sprachen eingesetzt und führt dort unter anderem zu deren besonders kurzer Syntax.

```
datatype day = Mon | Tue | Wed | Thu | Fri | Sat | Sun

fun day off(Sun) = true
      | day off(Sat) = true
      | day off( ) = false    // everything else

fun reverse(nil) = nil
      | reverse(head::tail) = reverse(tail) @ [head]

fun rev(nil) = nil
      | rev(0::tail) = [0] @ [tail]
      | rev(head::tail) = rev(tail) @ [head]
```

Nondeterminism and Backtracking

Prolog oder andere logikorientierte Sprachen verwenden Logikbäume.

```
//A is TRUE if B or C or D is TRUE
A if B or
C or
D.

C if E and
F and
G.

D if I or
H.
```

Kann als Baum gesehen werden dessen Knoten mit UND oder ODER angehängt sind.
Bei einer UND Verzweigung müssen alle Zweige abgearbeitet werden. Bei ODER wird Zweig für Zweig betrachtet und bei Evaluierung zu FALSE wieder abgebaut (Backtracking: Variablen gelöscht,...), bis einer (falls überhaupt) zu TRUE evaluiert. Dies wird prozedurale Interpretation (Ablauf wie in einer prozeduralen Sprache) genannt.

Event Driven Computation

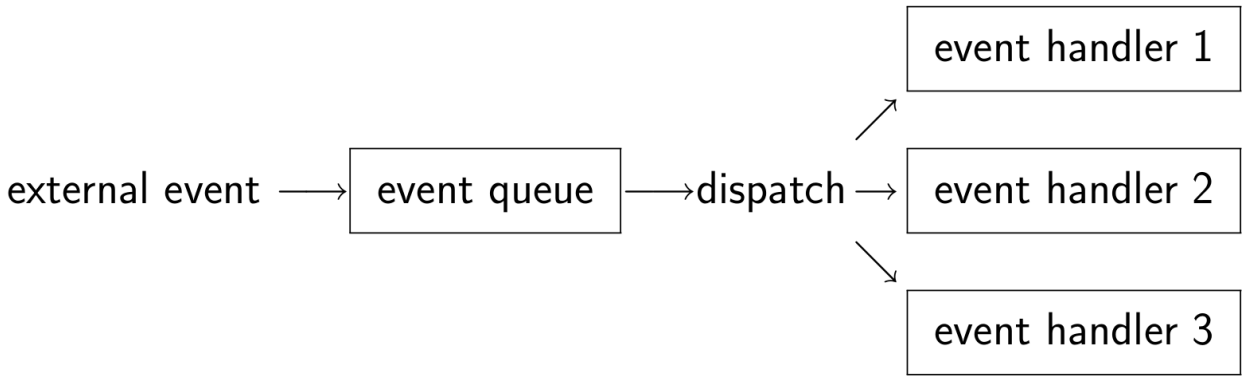
Viele Datenbanken unterstützen sogenannte Trigger, welche eine Event basierten Berechnung entsprechen.

```
on event                        //if “event” occurs
when condition                  //AND “condition” is satisfied
do action
```

```
on insert in EMPLOYEE
when TRUE
do emp number++
```

Passieren mehrere dieser Events „gleichtzeitig“, so gibt es keine vorgelegte Reihenfolge für deren Abarbeitung.

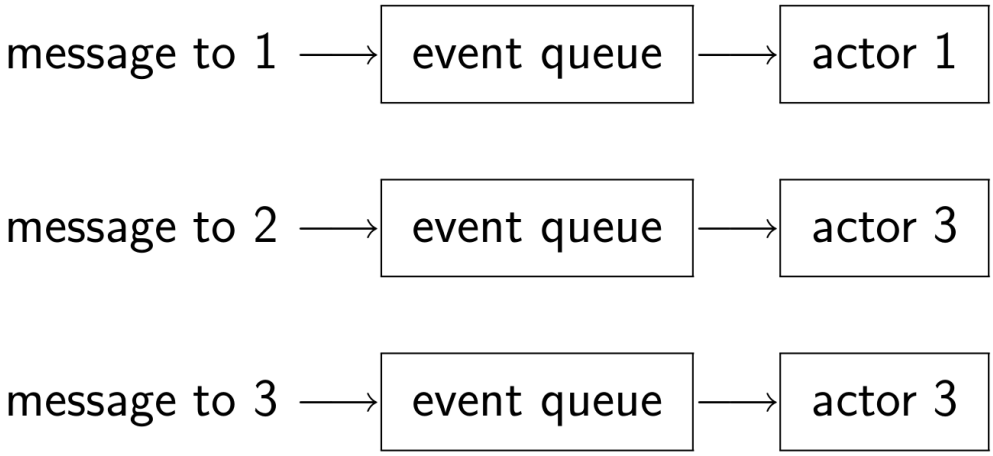
Event Driven Computation kommt weiters in manchen Betriebssystemen intensiv zum Einsatz (Zum Beispiel MS Windows).



Ein Ereignis wie zum Bespiel „Drücken der Linken Maustaste“ tritt auf. Dieses wird dann in einer event queue eingereiht. Ein Dispatcher nimmt die Ereignise aus der Queue und gibt diese an einen Handler, welcher auf ein derartiges Ereignis wartet weiter.

Scheme of Actors

Actors (aktive Objekte) sind das ursprüngliche Modell hinter der gesamten objektorientierten Programmierung.



Objekte schicken einander Nachrichten, welche vom Empfänger abgearbeitet werden. Jeder Actor hat eine Queue deren Inhalt er nach der Reihe abarbeitet. Die Länge dieser Queue ist jedoch problematisch.

Obwohl das Modell der passiven Aktoren lange Zeit die Oberhand hatte werden aktive Aktoren durch deren Überlegenheit in Sachen Nebenläufigkeit wieder populär.

Coroutines

Koroutinen sind eine Art „billige“ Nebenläufigkeit.

```
                          int i = 0;
unit client {             unit main {               unit next {
   int stop = ...;           resume next;              int step(){...};
      ...                 }                          ...
   while(i != stop) {                                 while(true) {
      ...                                                i += step();
   resume next;                                          resume client;
   }                                                  }
}                                                  }
```

3 Codesegmente:
main: startet das Programm
client: der Consumer (ließt ständig aus der globalen Variable)
next: der Producer (produziert ständig Daten und schreibt diese in die globale Variable)

Mit dem Aufruf von *resume target*, wird explizit angegeben, dass der Programmfluss in *target* fortgesetzt werden soll.

Eine aktuelle Sprache die dieses Konzept aufgreift ist Go (entwickelt von Google) mit seinen Go-Routinen.

Process Example: Producer and Consumer

```
process producer {                  process consumer {
      while (true) {                      while (true) {
            produce an element;                 take element from buffer;
            append it to the buffer;            operate on it;
      }                                   }
}                                   }
```

Buffer Operations

```
void append(int x) {
      count++;
      int i = next_in();
      buffer[i] = x;
}
```

```
int remove() {
      count--;
      int j = next_out();
      return buffer[j];
}
```

Bei echter Nebenläufigkeit können natürlich *append* und *remove* gleichzeitig aufgerufen werden: Es kommt zu mehreren Race-Conditions (der Wert von *count* könnte dadurch unvorhergesehen variieren,...) .
Um diese zu verhindern muss der Ablauf synchronisiert werden.

Declaration of Processes in Ada

Ada bietet die Möglichkeit sogenannte Tasks zu spezifizieren, welche im wesentlichen Prozessen entsprechen.

```
//task type → Typ eines Tasks
task type SERVER is
      entry NEXT REQUEST(NR: in REQUEST);
      entry SHUT DOWN;
end SERVER;
type SERVER PTR is access SERVER;         //Zeiger auf

MY SERVER: SERVER;                        //Instanz eines Task types = Prozess

//Direkte Spezifikation von Task (bereits eine Instanz)
task CHECKER is
      entry CHECK(T: in TEXT);
      entry CLOSE;
end CHECKER;

HIS SERVER PTR: SERVER PTR := new SERVER; //Erstellung auf Heap
```

Tasks dürfen allerdings nur sogenannte Entries exportieren. Diese gleichen syntaktisch herkömmlichen Prozeduren (können *in*, *out* oder *in out* Parameter haben).
Tasks können untereinander durch den Aufruf und vor allem die Parameter von Entries kommunizieren.
Ruft ein Task den Entry eines anderen auf, wird er suspendiert, bis der Aufgerufene den Entry abgearbeitet hat.

Synchronisierung

Semaphores

Semaphoren sind meist nicht direkt in Programmiersprachen umgesetzt, sondern werden über Bibliotheken integriert.
Semaphoren sind eine Art der Low-Level Synchronisierung. Sie sind im Wesentlichen nicht mehr als ein Zähler, welcher nicht kleiner als 0 werden kann (beim Erreichen von 0 wird der Prozess suspendiert). Der Wert des Zählers kann als Anzahl verfügbarer Ressourcen (Anzahl der möglichen Prozesse im kritischen Abschnitt) gesehen werden.
Semaphoren unterstützen die beiden Methoden:

P(semaphor): Erniedrigt den Wert des Semaphors um 1. Wird beim Eintritt in einen kritischen Abschnitt aufgerufen.

V(semaphor): Erhöht den Wert des Semaphors um 1. Wird beim Austreten aus einem kritischen Abschnitt aufgerufen.
Ist der Wert des Semaphor vor Aufruf der Operation 0, so wird ein eventuell darauf wartender Prozess (unbestimmt welcher, falls mehrere) aufgeweckt.

Beispiel:

```
var buffer buf;
semaphore mutex = 1;
semaphore in = 0;
semaphore spaces = buf.size();
process producer {
      int i;
      while(true) {
            produce(i);
            P(spaces);
            P(mutex);
            buf.append(i);
            V(mutex);
            V(in);
      }
}
```

```
process consumer {
      int j;
      while(true) {
            P(in);
            P(mutex);
            j = buf.remove();
            V(mutex);
            V(spaces);
            consume(i);
      }
}
```

Es werden 3 Semaphoren verwendet
mutex: die Kontrolle des kritischen Abschnitts
in: wieviel Elemente sind gerade im Buffer
spaces: wieviel freier Platz ist im Buffer
Der Producer produziert in einer Endlosschleife Werte, die er in den Buffer schreibt. Mit *P(spaces)* wartet er darauf, dass ein freier Platz (im Buffer) existiert. Mit *P(mutex)* wartet er auf exklusiven Zugriff auf den Buffer. Sind diese Bedingungen erfüllt, wird in den Speicher geschrieben und anschließend mit V*(mutex)* der Buffer wieder freigegeben. Mit *V(in)* wird der Wert des Semaphors, welcher die Anzahl der im Buffer vorhanden Elemente beschreibt, erhöht.
Der Consumer wartet mit *P(in)* genau auf diesen Semaphor. Danach sichert er sich analog dem Producer exklusiven Zugriff, nimmt ein Element aus dem Buffer und gibt den kritischen Abschnitt anschließend wieder frei. Mit *V(spaces)* wird der Semaphor für die Anzahl der freien Plätze um 1 erhöht.

Monitor

Erstmals verwendet in Concurrent Pascal (70er Jahre), repräsentiert ein Monitor einen kritischen Abschnitt. Ein Monitor ist folglich ein Modul (ein abstrakter Datentyp, eine Klasse), in dem die von Prozessen gemeinsam genutzten Daten und ihre Zugriffsprozeduren (oder Methoden) zu einer Einheit zusammengeführt sind. Zugriffsprozeduren mit kritischen Abschnitten auf den Daten werden als Monitor-Operationen speziell gekennzeichnet. Zugriffsprozeduren ohne kritische Abschnitte können vom Modul zusätzlich angeboten werden. Es ist garantiert, dass nur eine Prozedur welche zu einem Monitor gehört gleichzeitig aktiv sein kann (Monitor-Operationen werden unter wechselseitigem Ausschluss ausgeführt). Man kann sich den Monitor als einen Raum vorstellen in dem nur Platz für einen Akteur ist. Will ein weiterer Akteur in den Monitor, muss er warten (wird blockiert) biss der Monitor frei ist.
Bei Verwendung eines Monitors muss sich der Programmierer nicht mehr explizit (zum Beispiel durch den Einsatz von Synchronisationsprimitiven wie Semaphore) um Synchronisierung kümmern. Dadurch soll er sich besser auf die eigentlichen Schwierigkeiten des Programms konzentrieren können.

Using a Monitor in Concurrent Pascal

```
type fifostorage = monitor
      var   contents: array[1..n] of integer;
            total: 0..n; in, out: 1..n;
            sender, receiver: queue;
      procedure entry append (item: integer)
      begin if total = n then delay(sender);
            contents[in] := item;
            in := (in mod n) + 1; total := total + 1;
            continue(receiver)
      end;
      procedure entry remove (var item: integer)
      begin if total = 0 then delay(receiver);
            item := contents[out];
            out := (out mod n) + 1; total := total - 1;
            continue(sender)
      end;
begin total := 0; in := 1; out := 1 end
```

Variablen;
contents: Inhalt des Buffers
total: Anzahl der Elemente
in: wo kann das nächste Element geschrieben werden
out: wo kann das nächste Element gelesen werden
Der Typ queue ist eine Warteschleife (ähnlich Semaphor)
sender, *receiver*

Prozeduren:
Prozeduren im Monitor werden mit *entry* markiert.
append: Wenn der Buffer voll (*total = n*) ist, soll sich der Aufrufer in der Queue „s*ender“* einreihen (schlafen legen) und in dieser darauf warten dass er aufgerufen (aufgeweckt) wird.
Anschließend (eventuell erst nachdem man aufgeweckt wurde) wird in den Buffer geschrieben, der neue Index berechnet und die Anzahl der Elemente erhöht. Mit *continue(receiver)* wird jemand der in der „receiver“ Warteschlange ist aufgeweckt.
remove: Wartet auf die *receiver* und weckt in der *sender* queue.

Synchronization of Processes via Monitor

```
type fifostorage = ... see implementation above (Page 19) ...;

type producer = process (storage: fifostorage)
      var element: integer;
      begin cycle ...; storage.append(element); ... end end;

type consumer = process (storage: fifostorage)
      var datum: integer;
      begin cycle ...; storage.remove(datum); ... end end;

var   meproducer: producer;
      youconsumer: consumer;
      buffer: fifostorage;

begin meproducer(buffer);
      youconsumer(buffer);
end
```

Bei der Verwendung eines Monitors muss keine Rücksicht auf Synchronisierung genommen werden, da dies bereits im Monitor geschieht. Zuerst werden 2 Prozesse (consumer, producer) erstellt, die jeweils in einer Endlosschleife ihrem Handeln nachgehen. Dann werden Variablen von diesen, sowie ein Buffer erstellt. Anschließend werden die beiden Prozesse gestartet und ihnen der Buffer übergeben.

Using a Monitor in Java

```
public class IntBuffer100 {
      private int[] cont = new int[100];
      private int in = 0, out = 0, total = 0;

      public synchronized void append(int item) {
            while (total >= 100) try { wait(); }
            catch(InterruptedException e){}
            cont[in] = item; total++; if (++in >= 100) in = 0;
            notifyAll();
      }

      public synchronized int remove() {
            while (total <= 0) try { wait(); }
            catch(InterruptedException e){}
            int temp = cont[out]; total--; if(++out >= 100) out = 0;
            notifyAll();
            return temp;
            }
}
```

In Java wird der Monitor durch das Schlüsselwort *synchronized* eingeleitet. Das Warten auf eine Ressource muss in Java innerhalb einer Schleife erfolgen, da ein *notify* nicht die einzige Möglichkeit darstellt um aus einem *wait* aufgeweckt zu werden. Per Definition kann ein Aufwecken sogar ohne Grund erfolgen. Da nicht bekannt ist warum und von wem man aufgeweckt wurde muss die Bedingung innerhalb einer Schleife überprüft werden.
In Java muss keine Queue explizit angegeben werden, da jedes Objekt als Queue verwendet werden kann. In diesem Beispiel bildet das *this* Objekt die Queue.

NotifyAll(): Weckt alle wartenden Threads auf.
Notify(): Weckt nur einen Thread auf. Dies kann jedoch leicht in einen Stillstand führen, da ein Thread aufgeweckt werden kann, welcher sich in seiner Ausführung fehlerhaft verhält und versäumt seinerseits ein *notify* aufzurufen.

Rendezvous in Ada

In Ada wurde ursprünglich (vor der Einführung von Monitoren) das Rendezvous Konzept zur Synchronisierung verwendet. Dieses musste nicht wie bei anderen Sprachen oft üblich via Bibliotheken eingebunden werden, sondern war direkt in der Sprache enthalten.
In Ada verläuft hierbei die Kommunikation zweier Tasks asymmetrisch in dem Sinn, dass ein Unterschied besteht zwischen dem einen Entry aufrufenden und dem aufgerufenen Task. Entries sind jene Bestandteile, die man von außerhalb des Tasks aufrufen kann. Die Sprache definiert, dass immer nur genau ein Entry eines Tasks gleichzeitig aktiviert werden kann. Gleichzeitig eintreffende Aufrufe werden in einer Warteschlange gereiht.
Der Aufruf eines Task-Entry's unterscheidet sich syntaktisch nicht vom Aufruf einer Prozedur. Semantisch jedoch gibt es wesentliche Unterschiede. Abgesehen von der bereits erwähnten Warteschlange, in der die Rufer abgelegt und mittels der sie zufolge einer FIFO-Strategie abgearbeitet werden, wird der rufende Task in eine Wartezustand versetzt, wenn der gerufene Task nicht bereit ist, die Kommunikation sofort zu bewerkstelligen. Auf Seiten des gerufenen Tasks gibt es *Accept-Anweisungen* für die spezifizierten Entries. Falls ein solcher Task (der gerufene „Server“) zu einer Accept-Anweisung eines Entry's kommt, aber momentan kein Aufruf dieses Entry's vorliegt, so geht dieser Task in den Wartezustand über. Kann die Kommunikation ablaufen, so werden alle nach dem entsprechenden `accept` angegebenen Anweisungen abgearbeitet und anschließend beide Tasks fortgesetzt. Diese Art der Kommunikation von parallelen Prozessen nennt man *Rendezvous* (der aufrufende Task erbittet ein Rendezvous mit dem Aufgerufenen. Dieser kann das Rendezvous akzeptieren oder nicht).
Selective Wait erlaubt dem Server Task auf mehreren Entries gleichzeitig zu horchen.

Beispiel:

```
task Buffer_Handler is
      entry Append (I: Integer);
      entry Remove (I: out Integer);
end Buffer_Handler;
task body Buffer_Handler is
      Cont: array(1..100) of Integer;
      In, Out: Integer range 1..100 := 1;
      Total: Integer range 0..100 := 0;
begin loop select when Total < 100 =>
            accept Append(I: Integer) do Cont(In) := I end;
            In := (In mod 100) + 1; Total := Total + 1;
        or when Total > 0 =>
            accept Remove(I: out Integer) do I := Cont(Out) end;
            Out := (Out mod 100) + 1; Total := Total - 1;
      end select;
   end loop;
end Buffer_Handler;
```

Im *task body* Teil werden die Variablen deklariert, welche im vorherigen Beispiel im privaten Monitor Teil deklariert wurden.
Es folgt eine Endlosschleife mit einem *select*, welches beliebig viele Zweige (mit *or* getrennt) zulässt.
Mit dem *when* wird entschieden wann eine Nachricht akzeptiert wird. Ist *Total < 100* wird zum Beispiel nur *Append* akzeptiert. Der Befehl *accept* leitet das eigentliche Rendezvous ein. Der Aufrufer, welcher die Nachricht geschickt hat muss warten (blockiert) bis diese vom Server Task bearbeitet wurde.
Das *end* Statement beendet das Rendezvous.

Das Rendezvous System kann in Analogie zum Actor Model gesehen werden. Append und Remove sind jeweils Aktoren, die einander Nachrichten schicken. Eine Abweichung ist allerdings dass die Aktoren beim Rendezvous warten müssen bis die Nachricht die sie abgesendet haben bearbeitet wurde.

Using a Protected Type (Monitor) in Ada

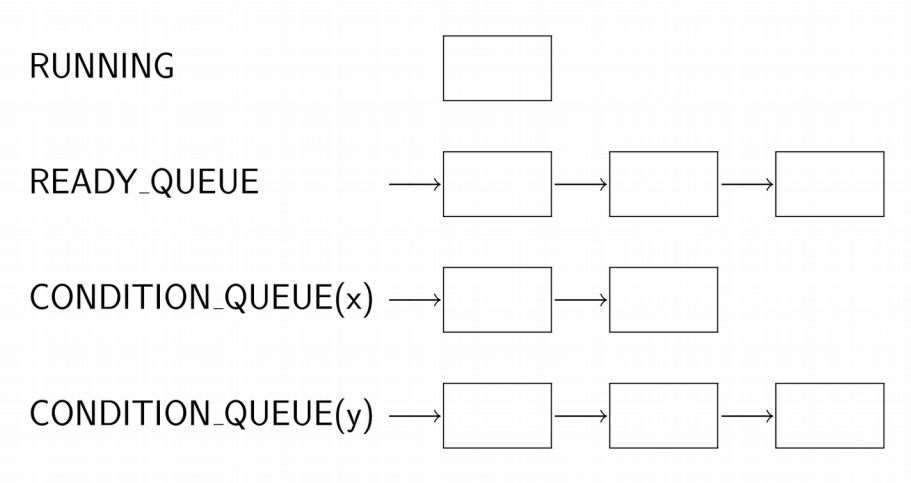
Ein Protected Type ist ein Record (~Klasse) welcher einen Monitor bildet. Diese wurden erstmals in Ada95 eingeführt.

```
//Definitionsteil
protected type Fifo_Storage is
      entry Append (Item: Integer);
      entry Remove (Item: out Integer);
private Contents: array(1..100) of Integer;
      In, Out: Integer range 1..100 := 1;
      Total: Integer range 0..100 := 0;
end Fifo_Storage;

//Deklarationsteil
protected body Fifo_Storage is
      entry Append (Item: Integer) when Total < 100 is begin
            Contents(In) := Item;
            In := (In mod 100) + 1; Total := Total + 1;
      end Append;
      entry Remove (Item: out Integer) when Total > 100 is begin
            Item := Contents(Out);
            Out := (Out mod 100) + 1; Total := Total - 1;
      end Remove;
end Fifo_Storage;
```

Im privaten Teil werden die Datenstrukturen definiert. Wie in Ada üblich wird der Definitions- vom Deklarationsteil getrennt.
Die beiden Methoden A*ppend* und R*emove* werden als E*ntries* implementiert.
Es muss in Ada kein wait oder notify geschrieben werden sondern lediglich die Bedingung unter welcher in den *Entry* eingetreten werden kann.
Eine derartige *when* Klausel, welche die Eintrittsbedingungen festlegt (auch Guard genannt), darf nur von Variablen (und Konstanten) abhängen die in dem Proteceted Type definiert wurden.
Ist die Eintrittsbedingung nicht erfüllt, wird der aufrufende Prozess in eine Warteschlange eingereiht. Nach Abarbeitung eines Entrys (oder Routine) werden alle Guards neu ausgewertet.

State Management by Operating System



Running: Ein Prozess der gerade lauft
READY_QUEUE: Prozesse die bereit dafür sind den Prozessor zugewiesen zu bekommen
Beliebig viele Condition Queues die Prozesse enthalten, welche auf ein bestimmtes Ereignis warten.
CONDITION_QUEUE(x): Prozesse warten auf Ereignis x.
CONDITION_QUEUE(y): Prozesse warten auf Ereignis y.

Operationen
enqueue:
dequeue:
empty:

Nach jedem Clock interupt wird folgende Operation ausgeführt:

```
Suspend and Select() {
      RUNNING = process_status;
      READY_QUEUE.enqueue(RUNNING);
      RUNNING = READY_QUEUE.dequeue();
      process_status = RUNNING;
}
```

Semaphor

```
Suspend_on_Condition(c) {
RUNNING = process_status;
CONDITION_QUEUE(c).enqueue(RUNNING);
RUNNING = READY_QUEUE.dequeue();
process_status = RUNNING;
}

Awaken(c) {
RUNNING = process_status;
READY_QUEUE.enqueue(RUNNING);
READY_QUEUE.enqueue(CONDITION_QUEUE(c).dequeue());
RUNNING = READY_QUEUE.dequeue();
process_status = RUNNING;
}
```

Monitor

“mutual exclusion” kann wenn dies zur Verfügung gestellt wird ganz einfach durch Ausschalten des Clock-Interrupts realisiert werden (momentaner Prozess wird nicht unterbrochen).

```
Continue(c) {
RUNNING = process status (interrupts enabled,
program counter set to return point from monitor call);
READY_QUEUE.enqueue(RUNNING);
if not CONDITION QUEUE(c).empty() {
READY_QUEUE.enqueue(CONDITION QUEUE(c).dequeue());
}
RUNNING = READY_QUEUE.dequeue();
process status = RUNNING;
}
```

Rendezvous

each “entry” has descriptor with these fields:

O: Boolean; true wenn entry offen ist. True wenn eine bestimmte Nachricht gerade akzeptiert wird. Dies ist der Fall wenn ein *select* Statement dafür existiert und dessen Bedienung erfüllt ist.
W: waiting queue (task descriptors of callers) : Hier warten alle Aufrufer
T: Deskriptor des Tasks zu dem der Entry gehört.
I: Zeiger auf die Instruktion des Accept Statements.

```
Call Entry(e) {
RUNNING = process_status;
DESCR(e).W.enqueue(RUNNING);
if DESCR(e).O {
      for all entries oe of DESCR(e).T do { oe.O = false; }
      RUNNING = DESCR(e).T;
      RUNNING.ip = DESCR(e).I;
} else { RUNNING = READY_QUEUE.dequeue(); }
process_status = RUNNING;
}
```

Beim Aufruf eines Entries wird der aktuelle Zustand gespeichert. Danach hängen wir uns in de Warteschlange des Empfängers.
Dann wird überprüft ob der Entry offen ist (darf man eintreten, o=true), ist dies der Fall wird eingetreten (nicht auf Aufforderung gewartet). Bevor wir den entry ausführen können müssen die *os* aller anderen Entries auf *false* gesetzt werden, dass niemand außer uns eintreten kann.
Wenn der Entry nicht offen war wird ein anderer laufbereiter Prozess von der Queue geholt.

```
At End Of Accept Body(e) {
RUNNING = process_status;
READY_QUEUE.enqueue(DESCR(e).W.dequeue());
READY_QUEUE.enqueue(RUNNING);
RUNNING = READY_QUEUE.dequeue();
process_status = RUNNING;
}

Execute Accept Statement(e) {
if DESCR(e).W.empty() {
DESCR(e).O = true;
DESCR(e).T = process_status;
RUNNING = READY_QUEUE.dequeue();
process_status = RUNNING;
} – – else simply continue executing the accept body
}



Execute Select Statement() {
LOE = list of open entries in selection;
if LOE is empty { raise Program Error; }
else {
if DESCR(e).W.empty() (for all e in LOE) {
for all e in LOE do {
DESCR(e).O = true;
DESCR(e).T = process_status;
}
RUNNING = READY_QUEUE.dequeue();
process_status = RUNNING;
} else { choose an e with not DESCR(e).W.empty(); }
}
proceed execution from instruction DESCR(e).I;
}
```