ABLAUFSTEUERUNG

Inhalt

Ablaufsteuerung

Sprünge — Ein vormals Emotions-beladenes Thema

Gotos considered harmful!

Die bisher behandelten Befehle
erlauben nur bestimmte Kontrollfluß-Muster

Das reicht nicht immer aus (praktisch gesehen)

Definition

Befehle zur Ablauf-Steuerung verändern den (normalen) Kontrollfluß

Arten der Ablaufsteuerung

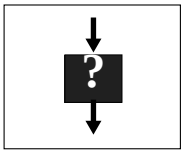
- Sprünge (*goto*)
- Ausgänge (*exit*, *return* usw.)
- Ausnahmen (*exception*)

Ausgangspunkt: *strukturiertes Programmieren*

Prinzip

Die bisher behandelten Befehle zum strukturierten Programmieren

- Zuweisung
- Prozeduraufruf
- Sequentielle Komposition
- bedingte Anweisung
- Schleifen



erlauben nur bestimmte Kontrollflußmuster:
ein Eingang – ein Ausgang

Jede Komposition mit solchen Befehlen hat auch
einen Eingang und einen Ausgang

Sprünge

Eine Erinnerung an die Assembler-Programmierung

Definition (Vereinbarung)

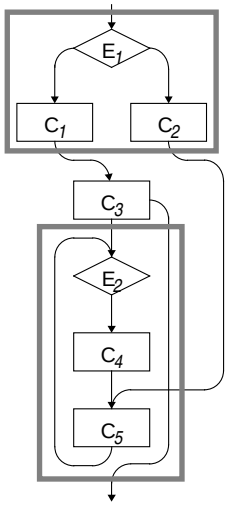
Programmstellen (Befehle) können markiert werden
L : C (L ist ein Bezeichner oder eine Zahl, C ist ein Befehl)

Benutzung

Zu solchen Marken kann gesprungen werden
goto L

Beispiel

```
begin
if E1 then C1
else
   begin C2;
   goto L
   end;
C3;
while E2 do
   begin
   C4;
L: C5
   end;
end
```



Markenvariable (pathologisch)

Abspeichern und Übergabe als Parameter

Markenfelder (*switch*)

Vorgänger der **case**-Anweisung

Sprünge und Blockstruktur

Das verträgt sich nicht sehr gut

Fortran, Algol:

Eine Marke gilt nur in dem Block, in dem die definiert wird

keine Sprünge von außen in einen Block

Pascal

Sprünge in Schleifen und Fallunterscheidungen hinein sind verboten

(das Beispiel ist illegal in Pascal!)

Beispiel: Sprung aus einer Prozedur

```
label 9;
procedure print (num, width: Natuaral);
   begin
   if width <=  0 then goto 9;
   if num    >= 10 then print(num div 10, width-1);
   write(chr(ord('0')+num mod 10))
   end;
begin
...
print (3,1);
print(1993,4);
...
9:
end.
```

ein Aufruf von `print` verläßt alle aktiven Instanzen der Prozedur

Was ist, wenn eine Marke in einer rekursiven Prozedur definiert wird?

welche Instanzen werden verlassen?

Ausgänge

Prinzip

Ein Programmteil wird (vorzeitig) verlassen

Das entspricht einem Sprung an das Ende dieses Programmteils

Vorteil gegenüber allgemeinen Sprüngen

- kein Rückwärtssprung — keine Probleme mit Termination
- Alle Befehle haben einen Eingang (aber evtl. mehrere Ausgänge)

Formen (Ada)

- Verlassen von Schleifen

  exit Name [**when** Expression]
- Rückkehr aus Prozeduren und Funktionen

  return [Expression]
- Übergrang zum nächsten Schleifendurchlauf
- Anhalten des Programms

  halt

Beenden von Schleifen

Form (in Ada)

Befehl im Rumpf einer Schleife

exit [Name [**when** *Boolean*_Expression]]

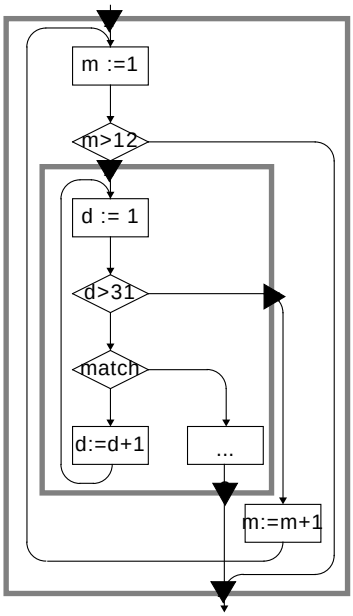
Bedeutung

verläßt die Schleife mit Namen Name (bzw. die innerste)

wenn die Bedingung erfüllt ist

Beispiel

```
search:
for m in Month loop
  for d in Day loop
    if matches
       (diary(m,d),i)
    then
      keydate:= (m, d);
      exit search;
    end if;
  end loop;
end loop;
```



nur lokale **exit**'s

mehrere **exit**'s möglich

nur Vorwärtssprünge

Rückkehr aus Prozeduren und Funktionen / Halten

Form (in Ada)

Befehl im Rumpf einer Prozedur und Funktion

return [Expression]

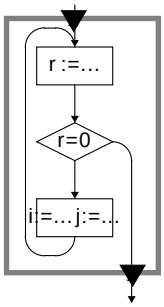
Bedeutung

verläßt die Prozedur bzw.

verläßt die Funktion mit dem Wert von Expression

Beispiel

```
function gcd (in i,j: Positive)return Positive is
   r: Positive;
begin
   loop
      r := i mod j;
      if r=0 then return j;
      end if;
      i := j; j := r;
   end loop;
end;
```



nur lokale **return**'s (im Rumpf)

mehrere **return**'s möglich

nur Vorwärts-Sprünge

Anhalten des Programms

halt

Ausnahmen

Ausnahmesituationen

- arithmetischer Überlauf
- unvollständige Ein-Ausgabe-Operation usw.

das prinzipielle Problem von Ausnahmesituationen

sie werden auf unteren Stufen des Programms hervorgerufen (geweckt)
und am besten auf höheren Stufen behandelt

Möglichkeiten der Behandlung von Ausnahmesituationen

Panik: Programm anhalten
das widerspricht der Forderung nach Robustheit

Do it Yourself

- Jede Funktion und Prozedur liefert einen *return code*, der aufgetretene Ausnahmen anzeigt.
- die Behandlung programmiert der Benutzer (wenn er es nicht vergißt)

das verlängert die Programme erheblich
und macht sie schwerer verständlich

Ausnahmen in Ada

Vereinbarung

```
Q_empty, Q-full: exception;
```

Ausnahmen können als Konstante
eines vordefinierten Aufzählungstyps betrachtet werden

Hervorrufen (Wecken, *to raise*)

```
raise Q_full;
```

An beliebiger Stelle im Programm

Behandlung

am Ende von Rümpfen und Blöcken

```
when
   Q_full  => ...
 | Q_empty =>...
 | others  =>...
```

Bedeutung

Hervorrufen einer Ausnahme
beendet die Ausführung des gerade aktiven Blockes

der aufrufende Block
behandelt die Ausnahme, oder
propagiert sie, und wird ebenfalls verlassen

Beispiel: Ausnahmen in Ada

Wetterdaten einlesen

```
procedure get_weather_data is
begin
   for m in Month loop
      begin
         get(rainfall(m));
      exception
         when data_error =>
            put("Invalid data for "); put(m);
            skip_data_item; rainfall(m):=0.0;
      end;
   end loop;
end get_weather_data;
```

`data_error` ist vordefiniert
und wird hervorgerufen von falsch aufgebauten `Float`-Literalen

nach Behandlung der Ausnahme im Schleifenrumpf
können weitere Werte eingelesen werden

Wetterdaten verarbeiten

```
procedure main is
begin
   get_weather_data;
   process_weather_data;
exception
   when end_error => put("Incomplete data");
end main;
```

`end_error` ist vordefiniert für unerwartetes Dateiende

nach Behandlung der Ausnahme wird die Prozedur `main` verlassen
(ohne das `process_weather_data` ausgeführt wird)

Übungen 8: Schleifenausgänge

Formuliere die Beispiele `search` so um, so daß kein **`exit`** benutzt wird.
Welche Version ist besser lesbar?

mit `exit`

```
search:
for m in Month loop
   for d in Day loop
      if matches
         (diary(m,d),i)
      then
         keydate:= (m, d);
         exit search;
      end if;
   end loop;
end loop;
```

ohne `exit`

```
declare
   m: Month := Month'first;
   d: Day := Day'first;
   not_found: Boolean := true;
begin
while not_found and then m <= Month'last loop
   while not_found and then d <= Day'last loop
      if matches (diary(m,d),i)
      then
         keydate:= (m, d);
         not_found := false;
      end if;
      d := d +1;
   end loop;
   m:= m'next;
end loop;
end
```

Übungen 8: `Queue_class` mit Ausnahmen (I)

Schreibe das Paket `queue_class` so um, daß für den Aufruf von

- `remove` bei leerer Schlange die Ausnahme `EmptyQueue`, und
- `append` bei voller Schlange die Ausnahme `FullQueue`

Schnittstelle

```
generic
   capacity : in Positive;
   type ITEM is private;
package queue_class is
   function is_empty return Boolean;
   function is_full return Boolean;
   procedure append (i: in  ITEM);
   procedure remove (i: out ITEM);
end queue_class;
```

Übungen 8: `Queue_class` mit Ausnahmen (II)

Implementierung

```
package body queue_class is
   FullQueue, EmptyQueue: exception;
   items : array (0..capacity-1) of ITEM;
   size, front, rear: Integer range -1..capacity-1;

   procedure is_empty is
   begin
      return size = 0;
   end is_empty;

   procedure is_full is
   begin
      return size = capacity;
   end is_full;

   procedure append (i: in ITEM) is
   begin
      if is_full then raise FullQueue;
```