Programmeren – Blok B

http://www.win.tue.nl/~wstomv/edu/2ip05/

College 6

Tom Verhoeff

Technische Universiteit Eindhoven
Faculteit Wiskunde en Informatica
Software Engineering & Technology

Opmerkingen aan T.Verhoeff@TUE.NL

Thema: Complexiteit beteugelen

Methode: Verdeel en heers

1. Probleem in deelproblemen splitsen
2. Deelproblemen afzonderlijk oplossen
3. Deeloplossingen tot totaaloplossing combineren

Mogelijke deelproblemen bij dobbelen

- De eventuele winnaar bepalen bij één ronde van het spel
- Inlezen van de worpen in een ronde
- Het maximum bepalen van een stel worpen
- Tellen hoe vaak een gegeven worp vóórkomt
- Bepalen wie een gegeven worp deed

Verdeel en heers in Pascal: Routines

- Routine = verpakte oplossing voor een (deel)probleem
- Twee soorten routines in Pascal:
  - **function** = Afkorting voor een formule
  - **procedure** = Afkorting voor een opdracht

Drie aspecten van routines

- Specificeren van deelproblemen: verdelen
- Maken van deeloplossingen: ...
- Gebruiken van deeloplossingen: heersen

Deze week alleen

- Specificaties *lezen* om routines te *gebruiken*

  (volgende keer: routines maken; specificaties bedenken)

Voorbeelden van Pascal procedures en functies

```
Read ( ... )

Write ( ... )

Randomize

Delete ( ... )
```

```
Random ( ... )

Length ( ... )

Pos ( ... )
```


Voorbeeld van Pascal procedure: specificatie

Syntax (formeel):

```
procedure Delete ( var S: String;
                   Index: Integer;
                   Count: Integer );
```

Semantiek (informeel, uit FreePascal RTL documentatie):

```
{ Delete removes Count characters from string S,
  starting at position Index.
  All characters after the deleted characters are
  shifted Count positions to the left, and
  the length of the string is adjusted. }
```


Voorbeeld van Pascal functie: specificatie

Syntax (formeel):

```
function Pos ( const Substr: String;
               const S: String ): Byte;
```

Semantiek (informeel, uit FreePascal RTL documentatie):

```
{ Pos returns the index of Substr in S, if S contains Substr.
  In case Substr isn't found, 0 is returned.
  The search is case-sensitive. }
```

Voorbeelden van Pascal routines: gebruik (aanroepen)

```
{ S = 'stomp' }

Delete ( S, Length ( S ), 1 )

{ S = 'stom' }
```

```
if Pos ( ' ', S ) = 0 then begin

  { S bevat geen spatie }

  ...

  end
```


Specificatie van routines: Syntax

function ⟨*funcnaam*⟩ ⟨*parameterlijst*⟩ : ⟨*typenaam*⟩ ;

procedure ⟨*procnaam*⟩ ⟨*parameterlijst*⟩ ;

waarbij ⟨*parameterlijst*⟩ leeg is, of bestaat uit

(⟨*parameters*⟩ ; . . . ; ⟨*parameters*⟩)

en ⟨*parameters*⟩ bestaat uit één van

const ⟨*paramnamen*⟩ : ⟨*typenaam*⟩	{ const parameter}
out ⟨*paramnamen*⟩ : ⟨*typenaam*⟩	{ out parameter }
var ⟨*paramnamen*⟩ : ⟨*typenaam*⟩	{ var parameter }
⟨*paramnamen*⟩ : ⟨*typenaam*⟩	{ value parameter }

Gebruik (aanroep) van routines: Syntax

⟨*funcnaam*⟩ ⟨*argumenten*⟩	{ te gebruiken als ⟨*formule*⟩ }
⟨*procnaam*⟩ ⟨*argumenten*⟩	{ te gebruiken als ⟨*opdracht*⟩ }

waarbij ⟨*argumenten*⟩ leeg is of bestaat uit

(⟨*argument*⟩ , . . . , ⟨*argument*⟩)

en ⟨*argument*⟩ bestaat uit

⟨*var*⟩ (een variabele) als in de specificatie **out** of **var** staat
en anders (bij **const** of niets) uit ⟨*formule*⟩ .

Type van argument moet passen bij type van parameter.

Soorten parameters: const, out, var, value

const parameter: invoer, via formule

out parameter: uitvoer, via variabele

var parameter: invoer en uitvoer, via variabele

value parameter: invoer, via formule; lokale kopie

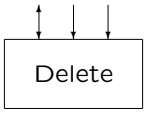
N.B. 1: variabele kan gebruikt worden als formule, niet omgekeerd

N.B. 2: verschil tussen **const** en value parameter zit in efficiëntie

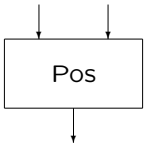
N.B. 3: **function** levert uitvoer als resultaat

Grafische weergave (gebruiken we i.h.a. niet)

```
procedure Delete ( var S: String; Index, Count: Integer);
```




```
function Pos ( const Substr: String; const S: String ): Byte;
```



Specificatie van routines: tweezijdige contracten

Contract fungeert als **interface** tussen

- **gebruiker** van oplossing (cliënt, aanroeper) en
- **maker** van oplossing (leverancier, implementator)

Contract legt vast *welk* probleem is opgelost,
d.m.v. **preconditie** (voorwaarde) en **postconditie** (gevolg).

Gebruiker hoeft niet te weten *hoe* het probleem is opgelost.

Maker hoeft niet te weten *waarvoor* de oplossing wordt gebruikt.

Contract: zorg en nut

		Contract		
		Preconditie		**Postconditie**
Partij	**Gebruiker**	zorg		nut
		↓		↑
	Maker	nut	→	zorg

Gebruiker zorgt dat preconditie geldt.

Maker benut deze preconditie en zorgt dat postconditie geldt.

Gebruiker benut deze postconditie.

Contractbreuk

- Als **gebruiker** het contract niet respecteert,
  dan treft maker geen blaam.

  Er kan dan van alles gebeuren . . . , garantie vervalt . . .
- Als **maker** het contract niet, maar gebruiker het wel respecteert,
  dan is de routine fout en treft gebruiker geen blaam.

  Geld terugvragen. . .

Specificatie van procedures: Semantiek van aanroep

procedure p (**const** c: $T1$; **out** v: $T2$; **var** x: $T3$; a: $T4$);
{ **pre:** $U(c, v, x, a)$
post: $V(c, v, x, a)$ }

met U, V predikaten over toestandruimte uitgebreid met parameters

Als vóór aanroep van p de **preconditie** $U(d, w, y, b)$ geldt,
dan geldt na aanroep van p de **postconditie** $V(d, w, y, b)$:

{@ $U(d, w, y, b)$ }
p (d, w, y, b)
{@ $V(d, w, y,$ old $b)$ }

Specificatie van procedures: Voorbeeld van aanroep

De standaardprocedure Inc heeft de volgende specificatie:

```
1 procedure Inc ( var i: Integer; d: Integer );
2   { Verhoog i met d
3     pre: True
4     post: i = old i + d  (old i = waarde van i bij aanroep)
5   }
```

Over de aanroep kunnen we als volgt redeneren:

```
1 var
2   n: Integer; { aantal gezochte waarden }
3
4 ...
5   {@ n = A }
6   Inc ( n, ord ( waarde = GEZOCHT ) )
7   {@ n = A + 1 als waarde = GEZOCHT en anders n = A }
```


Specificatie van functies: Semantiek van aanroep

function f (**const** c: $T1$; a: $T2$): $T0$;
{ **pre:** $U(c, a)$
ret: $F(c, a)$ }

met U een predikaat over toestandruimte uitgebreid met parameters
en F een formule over toestandsruimte uitgebreid met parameters

Als vóór aanroep van f de **preconditie** $U(d, b)$ geldt,
dan wordt na aanroep van $f(d, b)$ de waarde $F(d, b)$ geretourneerd.

Dobbelen (3): Aanhef en definities

```
1  program Dobbelen3;
2    { (c) 2001, Tom Verhoeff, Versie 3 }
3    { Lees 5 worpen en bepaal uitslag (werper van unieke maximum) }
4
5  const
6    NSpelers = 5; { aantal spelers, 1 <= NSpelers }
7    MaxWaarde = 12; { maximale waarde van een worp, 1 <= MaxWaarde
8
9  type
10   Speler = 1 .. NSpelers;  { de verzameling spelers }
11   Waarde = 1 .. MaxWaarde; { de verzameling worpwaarden }
12   WorpLijst = array [ Speler ] of Waarde; { worpen van een ronde
```

Dobbelen (3): Worpen inlezen

```
1 procedure LeesIn ( out wl: WorpLijst );
2
3   { pre:  True
4
5     post: Worpen ingelezen in wl[1..NSpelers] }
6
7   ...
```


Dobbelen (3): Bepaal hoogste worp

```
1 procedure BepaalMaxWorp ( const wl: WorpLijst;
2                           out m: Waarde );
3
4   { pre:  True
5
6     post: m = maximum van wl[1..NSpelers] }
7
8   ...
```


Dobbelen (3): Bepaal hoogste worp (alternatief)

```
1 function MaxWorp ( const wl: WorpLijst ): Waarde;
2
3   { pre:  True
4
5     ret: maximum van wl[1..NSpelers] }
6
7   ...
```


Dobbelen (3): Bepaal aantal keer dat worp voorkomt

```
1 procedure BepaalAantal ( const wl: WorpLijst;
2                          w: Waarde;
3                          out a: Integer );
4
5   { pre:  True
6
7     post: a = aantal keer dat w voorkomt in wl }
8
9   ...
```

(w hoeft niet de maximale worp te zijn, al is dat wel zo bij gebruik voor ons dobbelspel: dit heet generalisatie)

Dobbelen (3): Bepaal wie worp deed

```
1 procedure BepaalWerper ( const wl: WorpLijst;
2                          w: Waarde;
3                          out r: Speler );
4
5   { pre:  w komt voor in wl
6
7     post: wl [ r ] = w }
8
9   ...
```

(Specificatie biedt vrijheid als w vaker vóórkomt)

Dobbelen (3): Toestandsruimte

```
1 var
2   worp: WorpLijst;
3     { worp[i] = worp van speler i }
4   max: Waarde;
5     { maximum van worp[1..NSpelers] }
6   telMax: Integer;
7     { aantal keer dat max voorkomt in worp[1..NSpelers] }
8   winnaar: Speler;
9     { eventuele winnaar }
```


Dobbelen (3): Toestandsveranderingen

```
1  begin
2    LeesIn ( worp )
3  ; BepaalMaxWorp ( worp, max )
4  ; BepaalAantal ( worp, max, telMax )
5  ; if telMax = 1 then begin
6      BepaalWerper ( worp, max, winnaar )
7    ; writeln ( 'Speler ', winnaar, ' wint' )
8    end
9    else begin
10     writeln ( 'Geen winnaar' )
11   end
12   { de uitslag is afgedrukt }
13 end.
```


Specificatie van *FlipperkastNul*: aanhef

```
1  unit FlipperkastNul;
2    { (c) 2002, Tom Verhoeff, TUE }
3    { Spelregels: Start de flipperkast, voer een aantal Flips uit
4      en wijs de 0-bal aan. }
5
6  interface
7
8  const
9    AantalGaten = 10; { aantal gaten in de flipperkast }
10
11 type
12   GatNummer = 0 .. AantalGaten - 1; { de nummers van de gaten }
```

Specificatie van *FlipperkastNul*: *Start*

```
1  procedure Start ( const sNummer: Integer;
2                    out aantalBallen: Integer );
3
4    { Start de flipperkast en ontvang een aantal ballen. }
5
6    { pre:  Nog geen enkele procedure uit deze unit is aangeroepen.
7            sNummer = uw identiteitsnummer (de betaling :-).
8
9      post: aantalBallen = aantal ballen in de flipperkast.
10           1 <= aantalBallen <= AantalGaten.
11           De gaten 0 t/m aantalBallen - 1 bevatten elk een bal,
12           waarop een uniek getal van 0 t/m aantalBallen - 1. }
```


Specificatie van *FlipperkastNul*: *Flip*

```
1  procedure Flip ( const i, j: GatNummer );
2
3    { Voer een flip uit. }
4
5    { pre:  De flipperkast staat aan (Start is al aangeroepen).
6            i, j < aantalBallen (met aantalBallen uit Start).
7            Zij  a  het getal op de bal in gat i.
8            Zij  b  het getal op de bal in gat j.
9
10     post: Gat i bevat de bal met daarop de kleinste van a en b.
11           Gat j bevat de bal met daarop de grootste van a en b.
12           De ballen in de andere gaten zijn onaangeroerd. }
```


Specificatie van *FlipperkastNul*: *Stop*

```
1  procedure Stop ( const gatMetBal0: GatNummer );
2
3    { Wijs de 0-bal aan en stop de flipperkast. }
4
5    { pre:  De flipperkast staat aan (Start is al aangeroepen).
6            gatMetBal0 < aantalBallen (uit Start).
7            Gat gatMetBal0 bevat de bal met daarop getal 0.
8
9      post: Als antwoord correct is, dan is beloning uitgekeerd.
10           Programma is gestopt na lezen van <return>. }
```


Gebruik van *FlipperkastNul*

```
1  program FlipperenNul;
2    { Door Tom Verhoeff.  Illustreert gebruik van de flipperkast. }
3
4  uses
5    FlipperkastNul;
6
7  const
8    MijnNummer = 124866; { mijn studentnummer }
9
10 var
11   nBallen: Integer; { het aantal ballen }
12
13 begin
14   Start ( MijnNummer, nBallen )
15 ; Flip ( 1 , 2 )  (* Mogelijk geldt preconditie niet! *)
16 ; Flip ( 0 , 2 )  (* Mogelijk geldt preconditie niet! *)
17 ; Stop ( 1 )
18 end.
```

unit *Lists*;

```
1   { Unit om lijsten te manipuleren }
2
3  interface
4
5  const
6    MaxListLen = 24; { maximum lengte van een lijst, 1 <= MaxListLen }
7    MinEntry = 0; { kleinste waarde in een lijst }
8    MaxEntry = 99; { grootste waarde in een lijst, MinEntry <= MaxEntry }
9
10 type
11   TIndex = 0 .. MaxListLen - 1; { index in een lijst }
12   TEntry = MinEntry .. MaxEntry; { waarde in een lijst }
13   TList = record
14             len: 0 .. MaxListLen; { actuele lengte van de lijst }
15             val: array [ TIndex ] of TEntry; { rij van waarden in lijst }
16               { de lijst bestaat uit de waarden val[0] t/m val[len-1] }
17         end;
18   { Voor s: TList, moet s.val[i] gedefinieerd zijn voor 0 <= i < s.len }
```


Specificatie van *Generate* en *Find*

```
1  procedure GenerateList ( out s: TList; const n, m, d, c: Integer );
2    { genereer lijst s met lengte n volgens patroon m, d, c:
3      blokken van lengte d met gelijke waarden c, c+m, c+2m, ... }
4    { pre:  0 <= n <= MaxListLen,  0 < d,
5            MinEntry <= m * (i div d) + c <= MaxEntry  voor 0 <= i < n
6      post: s.len = n  en  s.val[i] = m * (i div d) + c  voor 0 <= i < n }
7
8  procedure Find ( const s: TList; const x: TEntry;
9                   out found: Boolean; out pos: TIndex );
10   { zoek x in lijst s }
11   { pre:  s is oplopend gesorteerd (duplicaten toegestaan)
12     post: found = 'er bestaat een i met 0<=i<s.len en s.val[i]=x', en
13           found impliceert  0 <= pos < s.len  en  s.val[pos] = x }
```


Afwegingen bij aanroepen van routines

- Procedure-aanroep is opdracht; functie-aanroep is formule
- Soorten parameters: **const**, **out**, **var**, value
- Gebruiker dient variabele te leveren voor **var** parameter
- Volgorde en types van parameters
- Denk aan het contract: welke parameters meegeven?
- Wat te doen met functieresultaat: "verbruiken" of opslaan?

Waarschuwing

Veel details rond routines zijn niet aangeroerd

en zijn ook niet van belang bij Programmeren – Blok B.

Gebruik alleen mogelijkheden die behandeld zijn en die je begrijpt.