Duijvestijn's geometrie in Bird's stijl uitgedrukt

Maarten Fokkinga, 8 april 1987

Voor de commissie HOP (Herstructurering Onderwijs in het Programmeren) van de INF-werkgroep WIP (Werkgroep Inleidend Programmeeronderwijs) heeft Duijvestijn een aantal teksten voorbereid met voorbeeld-programma's en voorbeeld-programma-ontwikkelingen in een functionele taal (Twentel). Hieronder geef ik in essentie exact dezelfde algoritmen, maar probeer dat te doen in de stijl van Bird's boek. Het resultaat vind ik verbluffend en onverwacht! Recursie blijkt niet meer expliciet nodig en ik kan volstaan met de notatie van Chapter 1-3 (respectievelijk Introduction, Data types, Lists). Het grote verschil in stijl tussen Bird's boek en Duijvestijn's programma's zit hem, mijns inziens, in het verschil tussen programmeren op functie-nivo (Bird) versus object-nivo (Dvst).

Het werken op functie-nivo vergt (misschien!) wat meer oefening/wat meer gewenning (of alleen maar enige ont-wenning??) dan het werken op object nivo, maar de resulterende teksten zijn m.i. véél duidelijker. De lezer oordele zelf.

Gebruikte notatie en hulpfuncties

Verwijzingen zoals [Dn] verwijzen naar de tekst van Dijkstra, pagina n.

Infix operator-identifiers zijn mogelijk in Miranda; Bird drukt ze in roman, ik onderstreep ze (en in Miranda moet er een $ voor).

Alle type-declaraties en typerings-declaraties zijn in feite overbodig; ik vermeld ze alleen ter documentatie (die bovendien door de compiler geverifieerd wordt).

Ik gebruik de volgende standaard functies uit Chapter 3 van Bird.

zip2 ([a, b, c ...], [x, y, z, ...]) = [(a,x), (b, y), (c, z), ...]
first2 (a, b) = a
snd2 (a, b) = b
last [a, b, c, ..., z] = z
head [a, b, c, ..., z] = a
iterate f x = [x, f x, (f·f) x, (f·f·f) x, ...]
map f [a, b, c, ...] = [f a, f b, f c, ...]
takewhile, dropwhile, all, some

Identifier-conventie: p ≈ point, ps ≈ points ≈ polygoon, t ≈ triangle, ts ≈ triangles

De programma's

```
point == (num, num)
line  == (point, point)        lijn opgespannen door 2 pnt
lineseg == (point, point)      begin en eindpunt
```

Een operator wrt (with respect to, ten opzichte van) die de ligging van een punt ten opzichte van een lijn of lijnstuk aangeeft. Een afgeleide operator is on; deze geeft aan of een punt op een lijn ligt. [D2]

```
(wrt) :: point → line → num
(on) :: point → line → bool
(x,y) wrt ((x1,y1),(x2,y2)) = (y-y1)*(x2-x1)-(x-x1)*(y2-y1)
p on l = (p wrt l = 0)
p rightof l = (p wrt l < 0)
p leftof l = (p wrt l > 0)
```

Polygonen, [D4]. Allereerst twee representaties; representatie polygon' wordt verderop niet meer gebruikt.

```
polygon == [point]
polygon' == ([num], [num])
polygonToPolygon' ps = (map first2 ps, map snd2 ps)
polygon'ToPolygon poly = zip2 poly
```

De functie 'binnen', [D 6, 8-10]. Ik geef deze als een operator inside3 die alleen correct is voor driehoeken (net als bij Duijvestijn). De algemenere operator inside maakt gebruik van de opsplitsing van een polygoon in driehoeken; dit wordt zo dadelijk pas behandeld. Merk op dat als ps de puntenrij van een polygoon is, dan zip2 (ps, tail ps ++ [head ps]) de gerichte lijnsegmentenrij van het polygoon is met, per afspraak, het binnengebied steeds rechts van de gerichte lijnsegmenten.

```
(inside3) :: point → polygon → bool
p inside3 ps = all [p rightof l | l ← zip2 (ps, tail ps ++ [head ps])]
                                                    || , if #ps = 3
```

Het opsplitsen van polygonen in driehoeken, [D 10-18]. Het probleem is om een polygon in disjuncte driehoeken te ontbinden zo dat de vereniging van (de gebieden van) de driehoeken precies (het gebied van) de polygoon is. De polygoon hoeft niet convex te zijn, maar we veronderstellen wel dat de omtreklijn zichzelf niet snijdt.

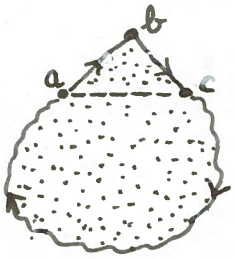
```
triangle == polygon                    || ter lengte 3
```

De idee is om een rij polygonen $ps_0, ps_1, ps_2, \ldots$ te definiëren, uitgaande van de gegeven polygoon $ps = ps_0$, waarbij de i-de ontstaat uit de (i-1)-de door toepassing van een functie 'cut' die één driehoek van ps_{i-1} afknipt en ps_i oplevert. Zodra voor zekere i geldt $\#ps_i = 2$, is het verwolg van de rij niet meer interessant (en zal wellicht $\perp$ zijn); het interessante deel van de rij verkrijgen we door toepassing van de functie (takewhile nontrivial) met nontrivial ps = #ps > 2.

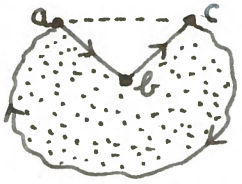
Uiteindelijk moeten we echter de afgeknipte driehoeken opleveren. Dus we breiden bovenstaand idee iets uit: in plaats van een rij $ps_0, ps_1, ps_2, \ldots$ vormen we een rij $(ts_0, ps_0), (ts_1, ps_1), (ts_2, ps_2), \ldots$ waarbij iedere ts_i een stel driehoeken is. Voor iedere i geldt: de driehoeken in ts_i en het polygoon ps_i zijn onderling disjunct en overdekken tesamen precies $ps = ps_0$. Met andere woorden, de ts_i vormen de afgeknipte driehoeken; $ts_0 = [\,]$. De gevraagde functie decompose kan nu als volgt gedefinieerd worden:

```
decompose ps =
    (first2 . last . takewhile psNontrivial . iterate cut')([], ps)
psNontrivial (ts, ps) = #ps > 2
cut' (ts, ps) = (t:ts, ps')  where (t, ps') = cut ps
```

Er rest ons nog de functie cut te definiëren die een driehoek t van een polygoon ps afknipt en het paar (t, ps') oplevert (zo dat " t ∪ ps' = ps ").
Beschouw nu de volgende polygonen:



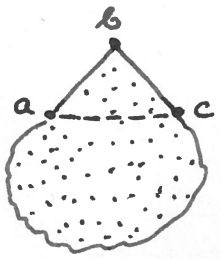
Δabc mag afgeknipt worden



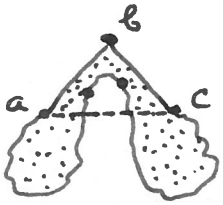
Δabc mag NIET "afgeknipt" worden

In de linker polygoon mag driehoek abc afgeknipt worden. In de rechter polygoon overdekt driehoek abc niets van de polygoon en mag (en kan) dus niet afgeknipt worden. Wanneer we een driehoek abc willen afknippen, en dus punt b uit de polygoon willen weglaten, dan moet de linkersituatie gelden: b ~~moet~~ mag niet rechts van ac liggen.

Een volgende voorwaarde waaraan driehoek abc moet voldoen om afgeknipt te mogen/kunnen worden, wordt geïllustreerd in de volgende figuren:



Δabc mag afgeknipt worden



Δabc mag niet afgeknipt worden

In de linkerfiguur behoort het gebied van Δabc geheel tot de polygoon; rechts is dat niet het geval. de voorwaarde luidt dus dat géén punt van het pologgoon binnen de driehoek abc mag liggen.

Tenslotte rijst de vraag welk punt van de polygoon aan bovenstaande twee voorwaarden voldoet. We weten dat als de polygoon niet-triviaal is, er zeker zo'n punt is -- dit is gegeven. Welnu, zij $ps = [a, b, c, d, e, \ldots, z]$ dan nemen we van de rij

$[a, b, c, \ldots, z]$, $[b, c, \ldots, z, a]$, $[c, \ldots, z, a, b]$, ...

de-eerste-de-beste waarvan de drie eerste punten aan de voorwaarden voldoen.

Aldus vinden we voor de functie cut:

```
cut = split • head • dropwhile ((~).ok) • iterate rotate
```

waarbij

```
rotate (x:xs) = xs ++ [x]
ok (a:b:c:ps) = ~ (b rightof (a,c))  ∧
                all [~ (p inside3 [a,b,c]) | p ← ps]
split (a:b:c:ps) = ([a,b,c], a:c:ps)
```

Hiermee is de programma-ontwikkeling voltooid. Eén opmerking kan echter nog wel gemaakt worden. Een afgesplitste driehoek kan eventueel een oppervlakte nul hebben, nml. wanneer de punten a, b en c op één lijn liggen (b binnen a,c of erbuiten). Wanneer dit niet gewenst is, dan kunnen we vooraan in de definitie van decompose nog de functie filter ((0 ≠).area) toevoegen:

```
decompose ps = ( filter ((0 ≠).area) • ----- ) ([], ps)
```

Deze toevoeging is voor het volgende niet nodig.

<u>Een functie 'binnen' voor polygonen</u>, [D 19]. Met behulp van de functie decompose luidt de uitwerking hiervan als volgt.

```
(inside):: point → polygon → bool
p inside ps = some [p inside t | t ← decompose ps]
```

* * *

Conclusies

De programmateksten die ik heb gepresenteerd zijn (of: vind ik) veel duidelijker en eleganter dan die Duijvestijn heeft gepresenteerd. Door de duidelijkheid, elegantie en eenvoudigheid heb ik er ook veel meer vertrouwen in: zo ze al niet 100% correct zijn, dan is een fout makkelijker te ontmaskeren. De vraag rijst nu waar die duidelijkheid etc aan te danken is. Ik zie daarvoor de volgende redenen.

1. Persoonlijke schrijfstijl, (o.a. keuze v.d. identifiers).
2. Duijvestijn's teksten waren bedoeld als allereerste college, waarbij bijvoorbeeld de lijst-comprehensie (verzamelingsnotatie) nog niet gebruikt werd. Ik heb daarentegen de eerste drie hoofdstukken van Bird bekend verondersteld.
3. Duijvestijn's programma's spelen zich veel meer af op het object-niveau: expliciete manipulatie van de lijst-elementen. Mijn programma's

zijn meer gericht op het functie-nivo: nieuwe functies (die op lijsten werken!) worden gedefinieerd als compositie van andere functies (die ieder ook op lijsten werken).

4. Samenhangend met het vorige punt: in Duijvestijns teksten wordt recursie veelvuldig expliciet gebruikt; bij mijn teksten is dat nergens het geval. Op zich ben ik niet tegen recursie, en vind ik recursie ook niet moeilijk; maar toch vind ik de recursieloze programma's --hier althans-- verreweg te prefereren boven die mét recursie.

Duijvestijn heeft gezegd steeds direkt te willen opschrijven wat bij hem opkwam, ("de ingenieursaanpak" - in tegenstelling tot de meer "academische aanpak" van Bird). Misschien is het zo dat de ietwat abstractere aanpak van Bird meer denkwerk vereist (alhoewel ik daar niet zeker van ben!); het resultaat is mijns inziens die moeite wel waard.

Uit al het bovenstaande trek ik de volgende twee conclusies:

1. Het boek van Bird is een serieuze handi-

daar om bij het eerste jaars onderwijs gebruikt te worden (zo het al niet de enige kandidaat is).

2. De geometrische problemen die door Duijvestijn behandeld zijn, zijn op zich niet al te moeilijk. Het zijn geschikte kandidaat-problemen voor een praktikum functioneel programmeren -- wanneer de eerste drie hoofdstukken van Bird behandeld zijn.