Practicum Programmeerprincipes 2006-2007

cderoove@vub.ac.be

REEKS 3

WERKEN MET FUNCTIES IN PICO

Functies, controlestructuren, recursie, iteratie, blokstructuren en functionele parameters in Pico.

Het woordenboek onder functietoepassing

Verklaar volgend transcript voor je aan de oefeningen begint!

```
> z : 10
10
> f(x) : z + x
<function f>
> f(3)
13
> z := 5
5
> f(3)
8
>
```

Functies met controlestructuren

Oefening 1. Definieer een functie `sign` die een getal als argument neemt en 1 teruggeeft als dat nummer positief is, -1 als het negatief is en 0 als het getal gelijk is aan 0.

Oefening 2. Definieer een functie `divides` die aangeeft of een getal exact een ander getal deelt.

Oefening 3. Definieer een functie `leap_year` die een jaartal als argument neemt en teruggeeft of het betreffende jaar een schrikkeljaar is. Alle jaren deelbaar door 400 zijn schrikkeljaren, alle jaren deelbaar door 100 (maar niet door 400) zijn geen schrikkeljaren en alle jaren die deelbaar zijn door 4 (maar niet door 100) zijn wel schrikkeljaren. Alle jaren die niet aan de voorgaande definitie voldoen zijn dan weer geen schrikkeljaren. Maak gebruik van de functie die je in de vorige oefening gedefinieerd hebt.

Recursie en iteratie

Oefening 4. Voorspel de uitvoer van de expressies `count1(4)` en `count2(4)` als deze functies als volgt gedefinieerd zijn:

```
count1(x) : if(x=0, display(x), {display(x); count1(x-1)});

count2(x) : if(x=0, display(x), {count2(x-1); display(x)});
```

Oefening 5. Definieer optelling en aftrekking aan de hand van de functie `dec` en `inc` die hun argument respectievelijk verminderen met 1 en met 1 vermeerderen. Je mag alleen `+` en `-` gebruiken in de definitie van `dec` en `inc`.

Oefening 6. Definieer de booleaanse functies `even(x)` en `odd(x)` recursief in functie van elkaar. Hint: bij mutueel recursieve functies zal je de eerste functie eerst een dummy voorschrift moeten geven en pas na de definitie van de tweede functie het correcte voorschrift kunnen toewijzen.

Oefening 7. Implementeer Ackermann's functie:

$$A(x,y) \equiv \begin{cases} y+1 & \text{if } x = 0 \\ A(x-1,1) & \text{if } y = 0 \\ A(x-1,A(x,y-1)) & \text{otherwise.} \end{cases}$$

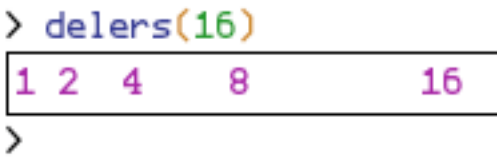
Oefening 8. Schrijf een functie `gcd` die de grootste gemene deler van twee getallen berekent volgens het algoritme van Euclides: gcd(a,0) = a en gcd(a,b) = gcd(b, a modulo b)

Oefening 9. We hebben al eens eerder een recursieve functie voor het verheffen van een getal tot een bepaalde macht geschreven. We kunnen de functie `power(x,y)` echter ook sneller bereken door gebruik te maken van `^2` en de pariteit (even/oneven) van het getal dat we verheffen. Implementeer `fast_power(x,y).` Kan je dit snelheidsverschil inschatten en verklaren? (tip: beschouw bij elke definitie hoe sterk het aantal recursieve functieoproepen groeit als je van het berekenen van een 2-de macht naar het berekenen van een 100-ste macht gaat).

Oefening 10. Schrijf een recursieve functie die een getal van het 10-delig talstelsel naar het binaire talstelsel omzet. De meest rechtste bit is 1 als het getal oneven is en 0 als het getal even is. Voor de tweede minst significante bit deel je het getal door 2 en je doet dezelfde test. Voor de derde minst significante bit deel je het vorige quotient opnieuw door 2 en ... Let erop dat je het binaire getal niet omgekeerd afdrukt!

Oefening 11. Schrijf een recursieve functie `toonTafel(n)` die de vermenigvuldigingstafel van het getal n afdrukt (zoals op de lagere school). Herschrijf de functie ook met een `while`-lus.

Oefening 12. Schrijf een functie `delers` die de delers van een getal print op een getallenas. Wanneer een getal bijvoorbeeld geen delers heeft tussen 4 en 8, moeten er 3 spaties tussen 4 en 8 geprint worden. Maak voor deze oefening gebruik van een `for`-lus.



Oefening 13. Schrijf een functie die de kleinste van 1 verschillende deler van een getal teruggeeft. Gebruik hiervoor een `until`-lus. Implementeer nu ook een priemgetaltest die gebruik maakt van je `smallest_divisor` functie en print, gebruik makende van een `for`-lus, de priemgetallen tussen 2 en 100. Later zullen we nog zien hoe je op een elegante wijze de zeef van Erasthostenes kan implementeren.

Oefening 14. Schrijf een functie `sum_integers(a,b)` die de som van alle gehele getallen tussen a en b berekent. Doe dit zowel iteratief als recursief.

Oefening 15. Schrijf de functies `sin(x,n)` en `cos(x,n) die` een sinus en cosinus benaderen aan de hand van de eerste `n` termen van de volgende reeksontwikkelingen. Gebruik hiervoor een `while`-lus.

$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

$$cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots$$

Het bereik van een variabele

Oefening 16. Beschouw het volgende Pico-programma. Voorspel de waarde van de expressie f(). Teken de structuur van het woordenboek waarbinnen expressies geëvalueerd worden voor de oproeping van f, bij de oproeping van f voor de definitie van g, bij de oproeping van g voor de definitie en oproeping van k, bij het einde van de oproeping van k, bij de oproeping van g na de oproeping van k, bij de oproeping van f na de oproeping van h.

```
{
  y : 6;
  x : 10;
  f() : { x : 5;
          z : y * 2 - 3;
          u : 0;
          g() : { u : 2;
                  v : u;
                  y := 3;
                  k() : { y : u;
                          x := y + 4
                        };
                  k()
                };
          h() : { x : y + u;
                  u := y + u;
                  z : y - x
                };
          g();
          h()
        }
}
```

Functies als teruggeven waarde

In Pico geven alle expressies bij evaluatie een waarde terug. Zo wordt bijvoorbeeld bij het definiëren of toewijzen van een functie deze functie telkens als waarde teruggegeven. Bij de evaluatie van een `begin`-expressie geeft deze eveneens altijd zijn laatste argument terug. We kunnen dus ook functies schrijven die bij de uitvoering andere functies als waarde teruggeven!

Oefening 17. Schrijf een functie `return_multiplier(n)` die een functie teruggeeft die zijn argument bij evaluatie steeds zal vermenigvuldigen met hetzelfde getal `n`:

```
> triple  : return_multiplier(3)
<function f>
> triple(2)
6
> double : return_multiplier(2)
<function f>
> double(2)
4
```

Functies als argumenten

Oefening 18. Schrijf een functie die de compositie van twee functies teruggeeft (deze neemt dus twee functies f(a) en g(b) als argument en geeft er de compositie compose(f,g) : f(g(x)) van terug).

Oefening 19. Schrijf analoog aan de recursieve definitie van de functie `sum_integers(a,b)` een recursieve functie `sum_squares(a,b)` die elk getal tussen a en b zal kwadrateren alvorens de som van de resultaten terug te geven: `sum_squares(5,7) = 25 + 36 + 49`

Oefening 20. Schrijf op dezelfde manier een recursieve functie `sum_every_two_integers` zodat
`sum_every_two_integers(5,9) = 5 + 7 + 9`

Oefening 21. Schrijf op dezelfde manier een functie `pisum(a,b)` die als volgt gedefinieerd is:

$$pisum(5,13) = \frac{1}{5*7} + \frac{1}{9*11} + \frac{1}{13*15}$$

$$pisum(1,\ldots) = \frac{1}{1*3} + \frac{1}{5*7} + \frac{1}{9*11} + \frac{1}{13*15} + \ldots = \frac{\Pi}{8} \text{ (Leibnitz)}$$

$$pisum(a,b) = \frac{1}{a*(a+2)} + \frac{1}{(a+4)*((a+4)+2)} + \frac{1}{(a+8)*((a+8)+2)} + \ldots$$

Oefening 22. Ondertussen heb je waarschijnlijk al het algemene patroon in deze oefeningen ontdekt:

$$\sum_{i=a,next(a),\ldots}^{b} term(i)$$

Het zou handig zijn om dit patroon uit te kunnen drukken in een functie `sum` die we steeds kunnen hergebruiken wanneer we ze nodig hebben. Deze functie is echter afhankelijk van twee andere functies:

- de functie `next(a)` die gegeven een getal, het volgende getal `i` zal teruggeven dat, na toepassing van `term` op i, bij de tot dan toe bekomen som geteld moet worden
- de functie `term(i)` die een bewerking op een getal uitvoert en wiens resultaat een term van de som zal vormen

Hiervoor moeten we functies echter op een manier gebruiken die we nog niet eerder gezien hebben: als argumenten van een functie. Dit is mogelijk omdat functies een van de 6 soorten van waarden zijn die we in Pico kunnen gebruiken: void, fractions, number, texts, functions en tables.

De definitie van deze functie is dan de volgende:

```
sum(term, next, a, b) : if(a > b,
                           0,
                           term(a) + sum(term,
                                         next,
                                         next(a), b))
```

We moeten ook nog een geschikte `term` en `next` functie definiëren. Om de som van de getallen in een interval te berekenen, maken we gebruik van de identiteitsfunctie `id(x)` voor `term` en een incrementfunctie `inc(x)` voor `next`:

```
id(x) : x;
inc(x) : x + 1;
```

We kunnen onze oude functie `sum_integers` dan als volgt definiëren in termen van de generieke sum functies:

```
sum_integers(a,b) : sum(id, inc, a, b)
```

Oefening 23. Druk nu zelf de functies `sum_squares`, `sum_every_two_integers` en `pi_sum` uit gebruik makende van de algemene functie `sum`.

Oefening 24. Zoals jullie weten, kan van vele functies het fixpunt benaderd worden door deze functie herhaaldelijk toe te passen. Implementeer een functie `fixpunt(epsilon, gok, f)` die het fixpunt van de functie `f` zal berekenen aan de hand van `.....f(f(f(f(f(gok)))))....` en zal stoppen van zodra het verschil tussen het resultaat van twee opeenvolgende functie-oproepen kleiner is dan epsilon.

Schrijf `fixpunt` op recursieve wijze. Aan de parameter `gok` geef je initieel een goede gok voor het fixpunt mee. In de recursiestap van je recursieve functie zal je vervolgens een nieuwe goede gok moeten berekenen vanuit de huidige gok. Denk ook goed na over de stopconditie van je recursieve functie.

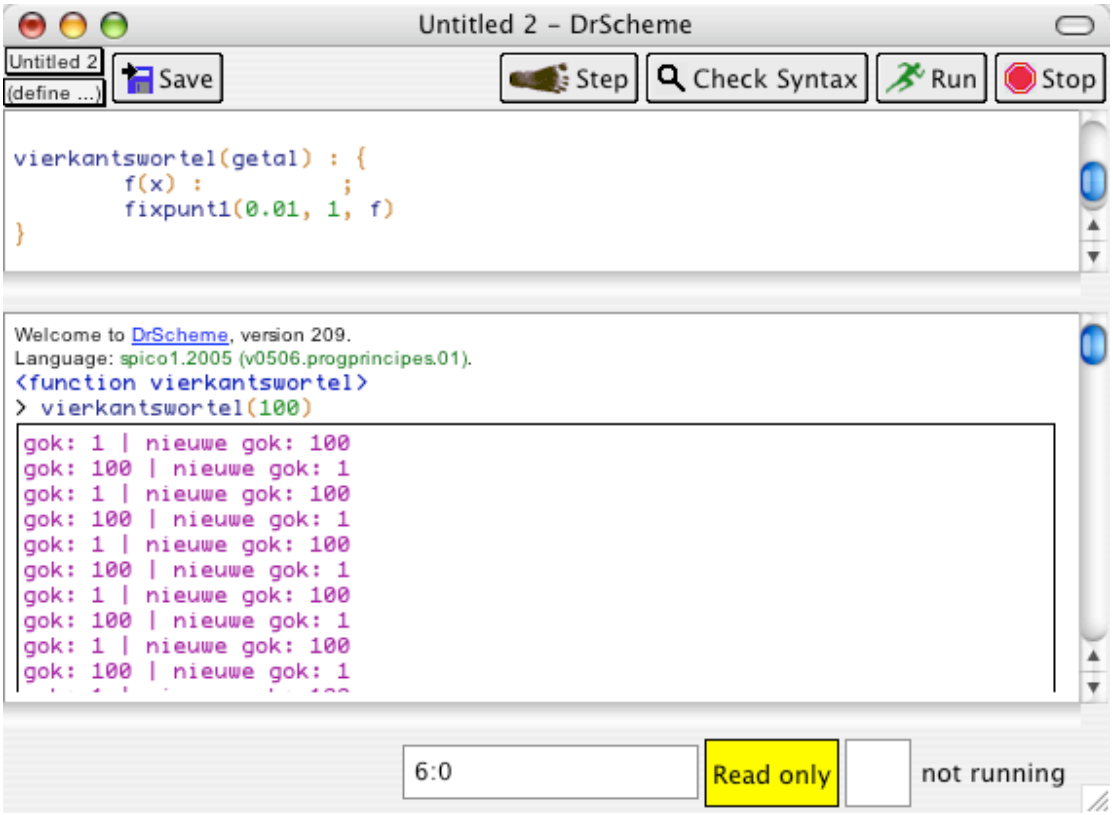
Je zou dan als volgt bijvoorbeeld het fixpunt van de cosinus-functie kunnen berekenen:

```
<function fixpunt>
> fixpunt(0.00001,1,cos)
0.739082
> cos(0.739082)
0.739087
>
```

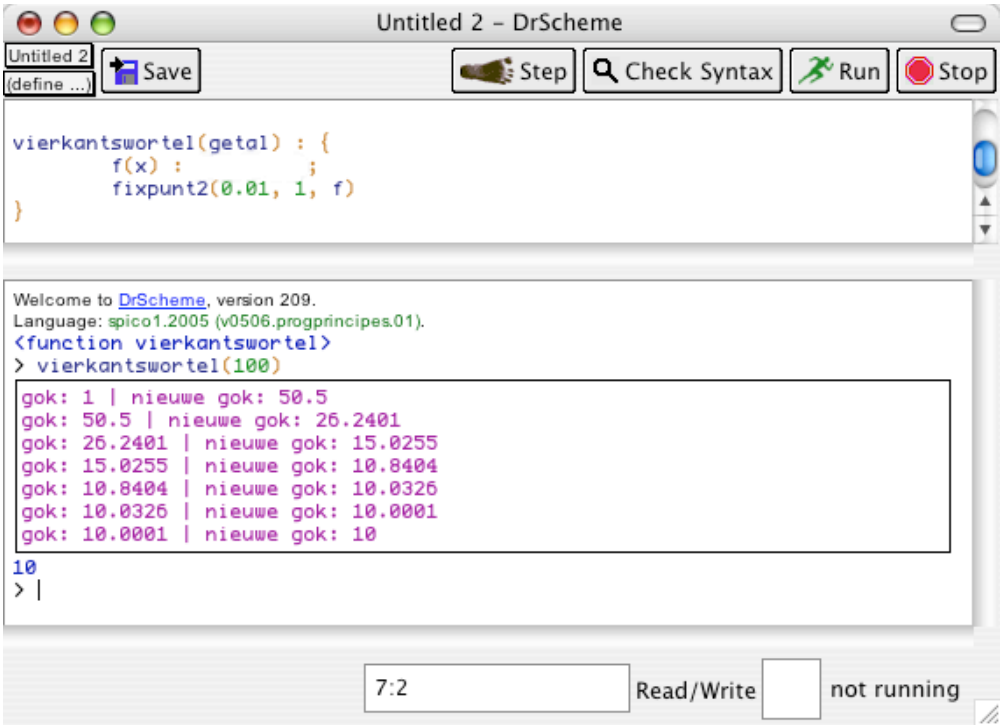
Oefening 25. Pas je `fixpunt`-functie aan zodat deze bij elke recursieve oproep zijn `gok` en `f(gok)` print. Noem deze functie `fixpunt1`. Voor de cosinus zou dit de volgende gegevens moeten printen:

```
<function fixpunt1>
> fixpunt1(0.01, 1, cos)
gok: 1 | nieuwe gok: 0.540302
gok: 0.540302 | nieuwe gok: 0.857553
gok: 0.857553 | nieuwe gok: 0.65429
gok: 0.65429 | nieuwe gok: 0.79348
gok: 0.79348 | nieuwe gok: 0.701369
gok: 0.701369 | nieuwe gok: 0.76396
gok: 0.76396 | nieuwe gok: 0.722102
gok: 0.722102 | nieuwe gok: 0.750418
gok: 0.750418 | nieuwe gok: 0.731404
gok: 0.731404 | nieuwe gok: 0.744237
gok: 0.744237 | nieuwe gok: 0.735605
0.735605
> |
```

Oefening 26. Probeer vervolgens een vierkantswortel te berekenen door het fixpunt te zoeken van een bepaalde functie. Wat zou het voorschrift van deze functie? Lukt dit met onze huidige `fixpunt1`-functie?



Oefening 27. Pas je `fixpunt1`-functie aan zodat oscillatiebewegingen gedempt worden door minder bruusk van een gok naar de volgende gok te springen:



Functionele parameters

Bij het gebruik van de algemene functie `sum(term, next, a, b)` is het wat omslachtig om telkens een nieuwe, extreem korte functie voor de `term` parameter (bijvoorbeeld `id(x) : x`) en de next parameter (bijvoorbeeld `inc(x) : x + 1`) te moeten schrijven. Gelukkig heeft Pico hier een handig hulpmiddel voor: functionele parameters. Dit zijn parameters van een functie die er zelf uitzien als een functie. Het volgende transcript zal het een en ander verduidelijken:

```
> f(a,g(x)) : g(1) + g(2) + g(a) + a
<function f>
> f(1,x*x)
7
```

Bij de definitie van een functie herkennen we functionele parameters omdat zij eruit zien als een ..... Als deze functie opgeroepen wordt, zullen de argumenten aan een parameter ..... geëvalueerd worden zoals dat bij een gewone parameter gebeurt. In de plaats daarvan, zal er binnen de functie een andere functie beschikbaar zijn met de naam van de functionele parameter en als implementatie het ..... dat aan de functionele parameter gegeven werd.

Oefening 28. Vergewis je ervan dat je ook de bindingen van de variabelen in dit uitgebreidere voorbeeld kan verklaren:

```
> x : 1000
1000
> f(x,a,g(x)) : x + a + g(a) + g(x)
<function f>
> f(3,50,x*x)
2562
> 3 + 50 + 2500 + 9
2562
```

Oefening 29. Pas de definitie van je compositie-operator uit een vorige oefening aan zodat deze werkt met functionele parameters. Let vooral op de naamgeving van de variabelen.

Oefening 30. Schrijf twee functies `average_damp(f(z))` en `derivative(f(z))` die elk een functionele parameter hebben en respectievelijk een nieuwe functie `averaged(x)` en `derived(x)` teruggeven. Het voorschrift van de terug te geven functies is als volgt:

$$averaged(x) = \frac{x + f(x)}{2}$$

$$derived(x) = \frac{f(x + dx) - f(x)}{dx}$$

Stel de globale variabele dx gelijk aan een erg klein getal (bijvoorbeeld `dx:0.0001` .. normaal gezien moet hier uiteraard een limiet berekend worden) en ga als volgt na dat derivative bij benadering werkt:

```
> d : derivative(z^3 + z^2 + z + 10)
<function derived>
> d(5)
86.0002
>
```