Theoretische Aspecten van Programmatuur

HC 1 (week 4)

Bob Diertens

Theory of Computer Science (TCS)

date: Februari, 27

version: February 24, 2017

Wat

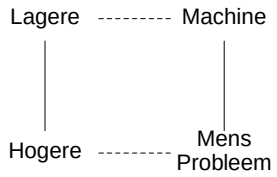
- Bestudering programmeer-constructies.
- Uitbreiding PGA (vnl. basis-instructies)
- Functies
- Berekeningsmodellen voor functies
- Concurrency support
- Communicatie
- Berekeningsmodel voor objecten
- Object-orientatie
- Gereedschap: Generalisatie & Abstractie

Hoe

- Colleges
- Practica
  - 2 opgaven per week
  - 1 verslag per week
  - inlevering:
    - programma's in platte tekstfiles
      + verslag (beknopt, geen proza)
    - tar of zip file (naam: <jouwnaam>-<opgave>.(tar|zip))
    - uiterlijk zaterdag 23:00?
- online (blackboard):
  - sheets: na betreffend college
  - prakticumopgave: na ieder college

Functies

Programmeertalen



Wat?

- maakt programmeren makkelijker
- verhoogt leesbaarheid

Hoe?

- betere / andere representatie (data)
- rij instructies vervangen door een / enkele / andere instructie(s) afkorting / constructie

Functie

Meest voorkomende constructie in programmeertalen.
(subroutine, procedure, ...)

Wat is een functie?

- programma-tekst
- evt voorzien van parameters

Hoe wordt een functie gebruikt?

- afkorting
  - macro-instructie
- abstractie
  - standaard
- generalisatie
  - vergroten toepasbaarheid

Afkorting

voorbeeld (PGLD)

```
    .                    ... def Add ...
    .                      .
    .                      .
+ x2 == Z;               Add;
#4;                        .
x1 = x1.s;                 .
x2 = x2.p;                 .
\#4;
  .
  .
  .
```

Add: tel x2 op bij x1

Abstractie

Wat?

Het benadrukken van het essentiele door weglaten van details.

Hoe?

Scheiding van
- betekenis van een functie (wat het doet)
- concrete tekst die de functie beschrijft (hoe het gedaan wordt)

In gebruik
- alleen wat het doet is van belang
- niet hoe het gedaan wordt

Dmv functie als programmeer-constructie krijgen we niveau's van abstractie.
Meer niveau's van abstractie dmv 'geneste' functies.

Abstractie (cont'd)

voorbeeld Add y2 to y1

```
x1 = y1;
x2 = y2;
Add;
y1 = x1;
```

Gebruik van functie als abstractie is beperkt.

We willen een functie waarbij we niet eerst de waarden op de juiste plaats moeten zetten.

Meerdere functies: Addx1x2, Addy1y2

Generalisatie

Parameterisering

- van verzameling specifieke afkortingen naar een generieke afkorting
- daarna is abstractie mogelijk

voorbeeld

```
Add(x1, x2);
Add(y1, x1);
Add(y1, y2);
```


Uitvoering

Algemeen

- interpreteren
- vertalen

  interpreteren

PGA Toolset

- simulatie - interpreteren
- projectie - vertalen

Uitvoering

Functies

- Projectie
  - expanderen
    (niet of beperkt recursief)
  - toevoegen mechanisme
    (stack)

    bv. PGLEcm $\rightarrow$ PGLEcr $\rightarrow$ PGLEc - mbv basis instructies
- Simulatie
  - mbv mechanisme zoals stack
  - expanderen wanneer nodig - herschrijven

Praktijk / Historie

- Macro Assembler expandering
- TRAC herschrijven
- LISP eval/apply

TRAC

Text Reckoning And Compiling

[Calvin N. Mooers, *TRAC, A Procedure-Describing Language for the Reactive Typewriter*, 1966]

- Teletypewriter (tty)
- evaluatie van een string dmv herschrijven
- meta character ’
- substrings `(...)`, `#(...)`, `##(...)`
- primitieven
- afkortingen (ds - define string)
  `#(ds, AA, ABA)`
  `#(cl, AA)`
  result: ABA
- parameters (ss - segment string)
  `#(ss, AA, B)`
  `#(cl, AA, C)`
  result: ACA
- idle string
  `#(ps, #(rs))`

TRAC (cont’d)

link: http://web.archive.org/web/20040925174226/http://tracfoundation.org/

voorbeeld

```
#(ds,append,XC)
#(ss,append,C,X)
#(ds,reverse,(#(ds,var,#(cc,X,()))#(eq,#(cl,var),(),(),
(#(cl,append,#(cl,var),#(cl,reverse,X))))))
#(ss,reverse,X)


#(ds, A, abc)
#(cl, reverse, A)
```



```
#(tn)#(cl, reverse, A)#(tf)’
*** TRACE cl: ‘reverse’, ‘A’?
*** TRACE cc: ‘A’, ‘’?
*** TRACE ds: ‘var’, ‘a’?
*** TRACE cl: ‘var’?
*** TRACE eq: ‘a’, ‘’, ‘’, ‘#(cl,append,#(cl,var),#(cl,reverse,A))’?
*** TRACE cl: ‘var’?
*** TRACE cl: ‘reverse’, ‘A’?
*** TRACE cc: ‘A’, ‘’?
*** TRACE ds: ‘var’, ‘b’?
*** TRACE cl: ‘var’?
*** TRACE eq: ‘b’, ‘’, ‘’, ‘#(cl,append,#(cl,var),#(cl,reverse,A))’?
*** TRACE cl: ‘var’?
*** TRACE cl: ‘reverse’, ‘A’?
*** TRACE cc: ‘A’, ‘’?
*** TRACE ds: ‘var’, ‘c’?
*** TRACE cl: ‘var’?
*** TRACE eq: ‘c’, ‘’, ‘’, ‘#(cl,append,#(cl,var),#(cl,reverse,A))’?
*** TRACE cl: ‘var’?
*** TRACE cl: ‘reverse’, ‘A’?
*** TRACE cc: ‘A’, ‘’?
*** TRACE ds: ‘var’, ‘’?
*** TRACE cl: ‘var’?
*** TRACE eq: ‘’, ‘’, ‘’, ‘#(cl,append,#(cl,var),#(cl,reverse,A))’?
*** TRACE cl: ‘append’, ‘c’, ‘’?
*** TRACE cl: ‘append’, ‘b’, ‘c’?
*** TRACE cl: ‘append’, ‘a’, ‘cb’?
*** TRACE tf: ?
cba
```

TRAC (cont'd)

Geheugen

- ruimte voor opslaan definities
- werkgeheugen
  - — groot array
  - — operaties voor het verschuiven van stukken array

LISP

LISt Processing

[John McCarthy, *Recursive Functions of Symbolic Expressions and Their Computation by Machine*, 1960]

- list structuur
- car, cdr, cons
- string -> list
- object lijst / associatie lijst en geheugen
- lambda
- evaluatie functies

  eval / apply complex werkend op een lijst
  - — geneste evaluatie dmv stack!
  - — parameters via associatie lijst (omgeving)

LISP (cont'd)

Symbolic Expressions

S-expressie:

- atom
- als e1 en e2 S-expr, dan ook (e1 . e2)

Lijst notatie (afkorting van S-expressie)

- (m) → (m . NIL)
- (m1 m2 m3) → (m1 . (m2 . (m3 . NIL)))
- (m1 m2 . x) → (m1 . (m2 . x))
- idem voor subexpressies

LISP (cont'd)

CAR CDR

(originele implementatie IBM 704 late vijftiger jaren)

Ondersteuning voor het splitsen van een 36bit word in 4 gedeeltes

- car (Contents of the Address part of Register number - 15 bits)
- cdr (Contents of the Decrement part of Register number - 15 bits)
- cpr (Contents of the Prefix part of Register number - 3 bits)
- ctr (Contents of the Tag part of Register number - 3 bits)

LISP (cont'd)

voorbeeld

```
(defun myappend (x y)
      (cond
         ((null x) y)
         (t (cons (car x) (myappend (cdr x) y)))
      )
)
(defun myreverse (x)
      (cond
         ((null x) nil)
         ((atom x) (cons x nil))
         (t (myappend (myreverse (cdr x)) (cons (car x) nil)))
      )
)


(myreverse '(A B C))
```

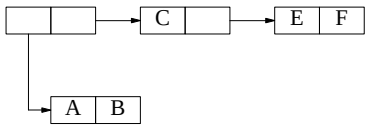


```
(trace myappend)
(trace myreverse)
(myreverse '(a b c))
1. Trace: (MYREVERSE '(A B C))
2. Trace: (MYREVERSE '(B C))
3. Trace: (MYREVERSE '(C))
4. Trace: (MYREVERSE 'NIL)
4. Trace: MYREVERSE ==> NIL
4. Trace: (MYAPPEND 'NIL '(C))
4. Trace: MYAPPEND ==> (C)
3. Trace: MYREVERSE ==> (C)
3. Trace: (MYAPPEND '(C) '(B))
4. Trace: (MYAPPEND 'NIL '(B))
4. Trace: MYAPPEND ==> (B)
3. Trace: MYAPPEND ==> (C B)
2. Trace: MYREVERSE ==> (C B)
2. Trace: (MYAPPEND '(C B) '(A))
3. Trace: (MYAPPEND '(B) '(A))
4. Trace: (MYAPPEND 'NIL '(A))
4. Trace: MYAPPEND ==> (A)
3. Trace: MYAPPEND ==> (B A)
2. Trace: MYAPPEND ==> (C B A)
1. Trace: MYREVERSE ==> (C B A)
(C B A)
```


LISP (cont'd)

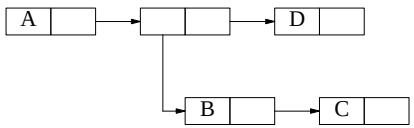
Geheugen

- associatie lijst - voor het opslaan van objecten
- werkgeheugen



((A . B) . (C . (E . F)))

A .. F - referenties aan associatie lijst



(A (B C) D)

Python

voorbeeld

```
def reverse(s):
     if len(s) == 0:
          return ""
     else:
          return reverse(s[1:]) + s[0]
print reverse('abc')
cba
def lreverse(s):
     if len(s) == 0:
          return []
     else:
          return lreverse(s[1:]) + [s[0]]
print lreverse([1, 2, 3, 4])
[4, 3, 2, 1]
def preverse(s):
     return s[::-1]
print preverse('abc')
cba
print preverse([1, 2, 3, 4])
[4, 3, 2, 1]
```

Python (cont'd)

Geheugen

???

Interpretatie

- TRAC, LISP en Python (en vele anderen) kunnen data interpreteren als code
- mechanisme: eval / apply complex
- mechanisme is (virtuele) machine

Nodig:

- ontleder (parser)
- opslag
- uitvoerder

Functies als Uitbreiding Taal

- De ene taal kan dit, maar de andere kan dat.
- Vaak is het verschil niet meer dan een bibliotheek van functies.

Het lijkt of deze functionaliteit behoort tot een taal.

Functie Uitbreidingen in PGA Toolset

- MPPV - MPP with values
- HMPPV - High-level MPPV
- MSP - Molecular Scripting Primitives
- MSPea - MSP with Eval/Apply complex

MPPV

`x.+f:t`
add value field f of type t (int (non-negative), bool) with default value (0, false)

`x.f = u`
assign value u
return false if value is not of the right type

`x.f = y`

`x = u`

`x = y`

`x == u`

`x == y`

HMPPV

- combines instructions
- new type: str (default "")
- extfocus (extended focus): a focus (x), field selection (x.f), compound field selection (x.f.g)
- instruction returns false when a field selection does not exist

HMPPV (cont'd)

`extfocus = new`

`extfocus1 = extfocus2`

`extfocus.+f`

`extfocus.+f = new`

`extfocus1.+f = extfocus2`

`extfocus.+f:t`

`extfocus.+f:t = u`

`extfocus1.+f:t = extfocus2`

`extfocus.-f`

`extfocus/f`

`extfocus1 == extfocus2`

`extfocus?`
true when extfocus is an atom, false otherwise

`extfocus?t`
true when extfocus is of type t, false otherwise

HMPPV to MPPV

x.f.+g:int = y.f.g in PGLEc.MPPV

```
+ y/f {;
      hy = y.f;
      + hy/g {;
            hy = hy.g;
            + x/f {;
                  hx = x.f;
                  - hx/g {;
                        hx.+g:int;
                        hx.g = hy;
                  };
            };
      };
}
```

+ x.f.+g:int = y.f.g {; ... } in PGLEc.MPPV

```
result = false;
+ y/f {;
      ...
                        hx.+g:int;
                        + hx.g = hy {;
                              result = true;
                        };
      ...
};
+ result == true {; ...  }
```

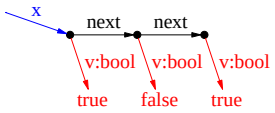
Values in MPP

Booleans

```
true = new;
false = new
```

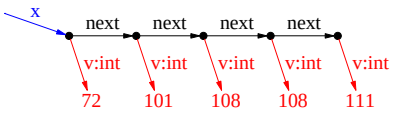
Naturals / Integers

representation in binary form by a list of booleans



Strings

a string is a list of characters and a character can be represented by a number



MSP

Operations on numbers:

```
incr extfocus
incr extfocus n
incr extfocus1 extfocus2
decr extfocus
decr extfocus n
decr extfocus1 extfocus2
```

Operations on strings:

`first extfocus1 extfocus2`
first character in string extfocus1 is assigned to extfocus2
return false if string is empty

`delfirst extfocus`
removes the first character from extfocus
return false if string is empty

`append extfocus1 extfocus2`
appends string in extfocus2 to the end of string in extfocus1

MSP (cont'd)

Type conversion:

`int extfocus1 extfocus2`
converts string in extfocus1 to int and assigns it to extfocus2

`str extfocus1 extfocus2`
converts integer in extfocus1 to a string and assign it to extfocus2

MSPea

Simulation of PGLA

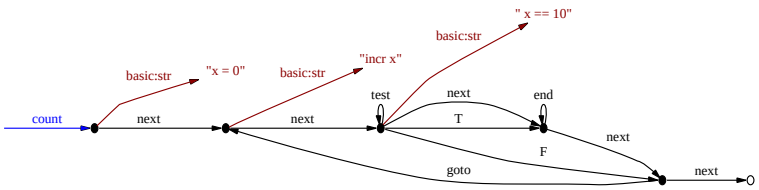
`compile extfocus`
compiles the string in extfocus into a molecule and assigns it to extfocus

`apply extfocus`
if extfocus is a string, it is executed as a basic instruction and returns the value returned by the basic instruction

`eval extfocus`
if extfocus is a string, it compiles the string into a molecule and evaluates the molecule
if extfocus is an atom, it is evaluated
returns the value of the last executed basic instruction

MSPea

```
count = "x = 0; incr x; + x == 10; !; \\\\#3";
compile count
```



De compile instructie kan uitgeprogrammeerd worden in PGLEc.MSP.

De eval instructie kan uitgeprogrammeerd worden in PGLEc.MSP.
(op basis van aanwezigheid apply instructie)

Samenvatting

1. Abstractie
2. Uitbreiding met data
3. Uitbreading functies op data
4. Uitbreiding eval / apply complex

CONTENTS