Lazy evaluation op expressie-nivo uitgelegd

Maarten Fokkinga, 15 aug. '85, herzien 2 dec. 1985

Lazy evaluation is een evaluatie-methode voor functionele programma's waarmee duplicaties van evaluatiestappen voorkomen worden. Wij geven hier een uiteenzetting van lazy evaluation geheel in termen die de praktiserend programmeur (in tegenstelling tot implementeur) gebruikt.

Met dank aan Mirjam Gerritsen, Gerrit van der Hoeven en Stef Joosten voor hun commentaar op een eerdere versie van dit verhaal.

***

Aan de hand van een voorbeeld zullen we lazy evaluation uitleggen. Als voorbeeld kiezen we de programmering van het n-de (tweede) priemgetal, door middel van een subscriptie op de lijst van alle priemgetallen, bepaald volgens de methode van Eratosthenes. De definities luiden als volgt. (We schrijven (f a) voor de toepassing van functie f op argument a, en net zo (f a b) voor f toegepast op a en b.)

```
zeef (x:X) = x: zeef {y | y <- X; x ∤ y}
    -- x ∤ y is kort voor: x deelt y niet, i.e. y mod x ≠ 0
from n = n: from (n+1)
priemL = zeef (from 2)
sub n (x:X) = if n=1 then x else sub (n-1) X
```

In het gezichtsveld van deze definities wordt het tweede priemgetal uitgedrukt door

```
sub 2 priemL.
```

Heel in het algemeen kunnen we ieder evaluatieproces omschrijven als de stapsgewijze transformatie van een expressie totdat een "gewenste vorm" verkregen is. We moeten daarbij wel bedenken dat expressies op de een of andere manier in de machine gerepresenteerd zijn; wij zullen echter nauwelijks acht slaan op die representatie en de schrijfwijze van de programmeur blijven volgen. De "gewenste vorm" die uiteindelijk na het evaluatieproces moet resulteren, bestaat uit getalnotaties (cijferrijen, eventueel voorzien van een teken), waarheidwaarden (true, false), characters, en (geneste) lijsten hiervan. Met name komen er geen identifiers en functietoepassingen in voor en geen operaties zoals +, -, *, =, $\wedge$, $\vee$, hd, tl. Een transformatie-stap bestaat uit de vervanging van een subexpressie door een andere volgens een van de definities of volgens de vaste betekenis van de standaardoperatoren, bijvoorbeeld

```
(3 + 4)       =>  7
from 7        =>  7: from (7+1)
from (3+4)    =>  (3+4): from ((3+4)+1)
```

Bij iedere vervangingsstap is de vervangen subexpressie zeker niet van de gewenste vorm (daarom wordt hij juist vervangen!) en de vervangende subexpressie mogelijk wel (maar niet altijd). Bij de "echte" functionele talen (zoals SASL, KRC, Miranda, Twentel, Hope, FP etc) doet het er nauwelijks toe wanneer welke subexpressie wordt vervangen: verschillen tussen strategieën uiten zich alleen in het al of niet bepaald zijn van de resultaten. Wanneer bij veschillende evaluatiestrategieën beide resultaten bepaald zijn, dan zijn ze ook gelijk. (Een taal zoals LISP of zelfs puur LISP mist deze eigenschap; daar is kennis van de gevolgde evaluatiestrategie dus van groot belang.)

Lazy evaluation is nu te omschrijven als de methode van evalueren (dwz de stapsgewijze transformatie uit te voeren) waarbij

(i) onderdelen van een subexpressie niet geevalueerd worden als dat nog niet nodig is om op de subexpressie een transformtiestap toe te passen, en

(ii) duplicaties van onderdelen van een subexpressie (t.g.v. een vervangingsstap) voorkomen worden door die onderdelen gedeeld (ge-"share"-d) te representeren.

We geven hiervan nu een voorbeeld. De gedeelde representatie van een subexpressie geven we aan door een voorkomen met een griekse letter te merken (als prefix-superscript) en op de andere voorkomens louter die letter te schrijven. Met een verticale pijl geven we aan welke subexpressie vervangen wordt: de pijl staat bij het eerste symbool van de te vervangen subexpressie en wijst naar de vervangende; de ongewijzigde delen schrijven we niet uit. Een pijl kan tijdelijk onderbroken worden, namelijk wanneer het eerst nodig is om onderdelen te evalueren. Lees ook de toelichting na Figuur 1.

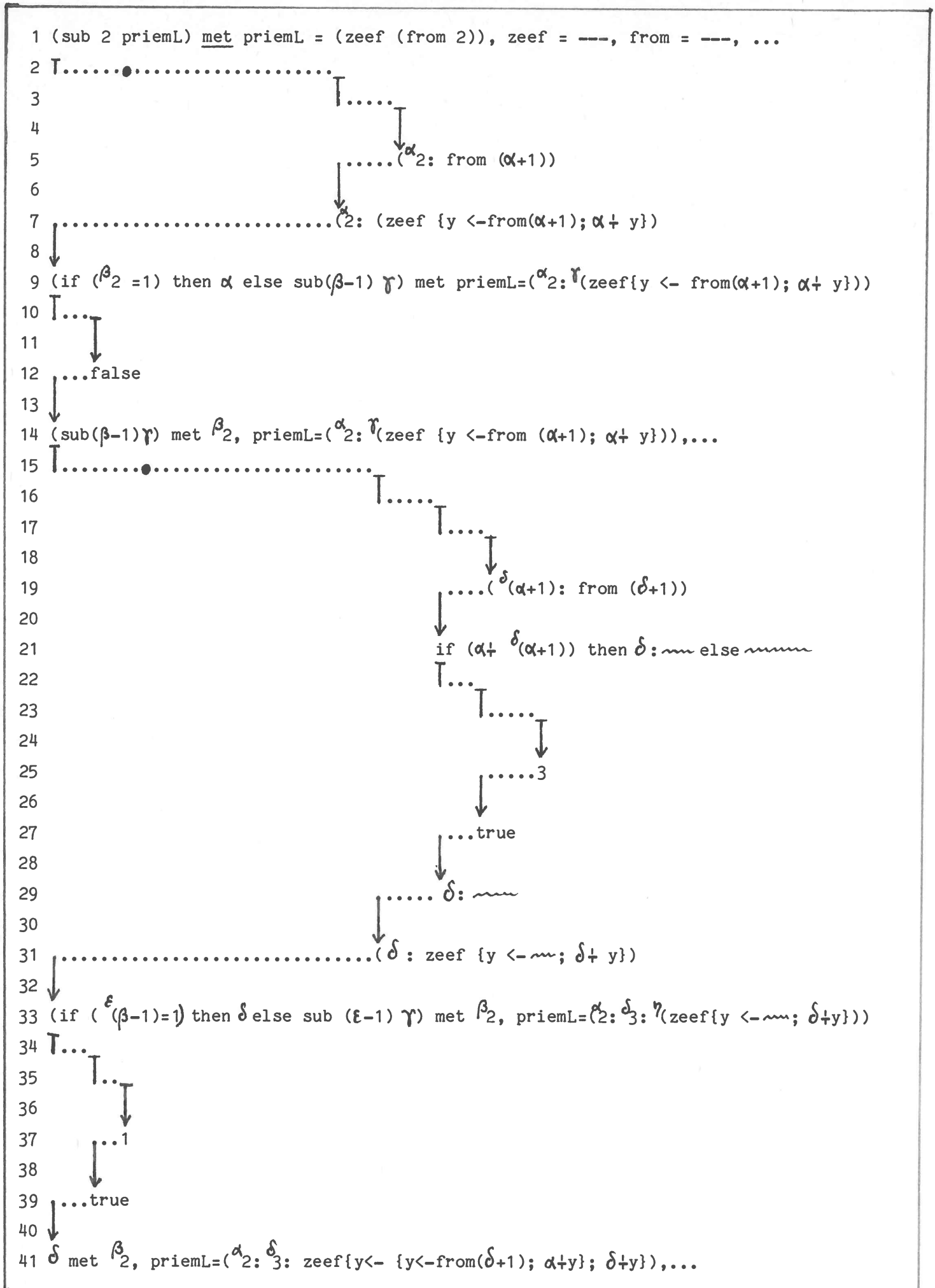
<zie nu Figuur 1>

Toelichting bij Figuur 1.

ad r.2 Vervanging van de expressie (sub 2 priemL) volgens de definitie van sub, nml. sub n (x:X) = ..., vereist dat eerst z'n tweede argumentexpressie tot de vorm (arg':arg") wordt geevalueerd. Het tweede argument is de identifier priemL, i.e. een verwijzing naar de definierende expressie elders.

ad r.3 De evaluatie van (zeef (from 2)) volgens de definitie van zeef, nml. zeef (x:X) = ..., vereist dat eerst de argumentexpressie tot de vorm (arg':arg") wordt geevalueerd.

ad r.4/5 De vervanging van (from 2) volgens de definitie from n= n: from (n+1) zou tot duplicatie van het argument kunnen leiden. Dit wordt vermeden door een voorkomen van dat argument met α te benoemen en op het andere voorkomen louter α en niet de argumentexpressie zelf te zetten. Overigens zou duplicatie van de expressie 2 niet erg zijn; erger is het als een nog niet volledig uit-geevalueerde expressie gedupliceerd zou worden (omdat dan ook vervangingsstappen gedupliceerd zouden worden en de evaluatie meer tijd zou vergen).



FIGUUR 1. LAZY EVALUATION;

(SUB 2 PRIEML) EVALUEERT TOT 3 MET ALS NEVENEFFEKT ENIGE EVALUATIE VAN PRIEML.

ad r.14 De vervanging van regel 9 door regel 14 heeft tot gevolg dat de met β benoemde subexpressie zou verdwijnen, terwijl β zelf niet verdwijnt. Uiteraard moeten we ergens blijven aangeven voor welke expressie β staat. Dat hebben we hier in regel 14 gedaan door β_2 in de lijst van definities op te nemen.

ad r.41 De vervanging van regel 33 door regel 41 heeft tot gevolg dat alle β's-als-subexpressie verdwijnen. Bijgevolg is het eigenlijk niet nodig de definitie β_2 nog langer te bewaren. Deze mag "als rommel opgeruimd worden" (garbage collection).

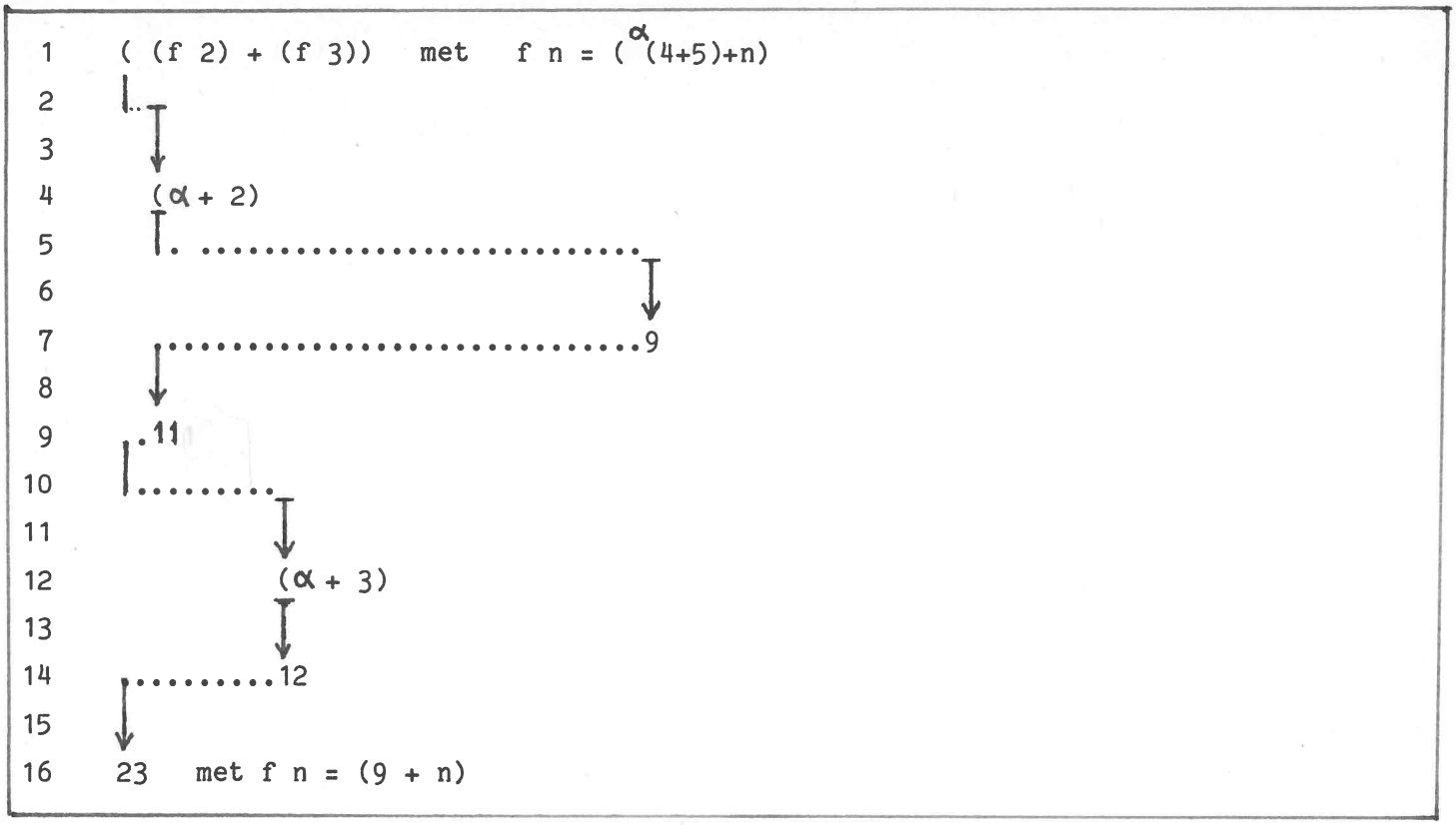
ad r.41 De totale evaluatie, volgens lazy evaluation, van de expressie (sub 2 priemL) kost 14 vervangingsstappen. Het resultaat is 3 terwijl als neveneffekt de oorspronkelijke definitie priemL = (zeef (from 2)) ook is geevalueerd (geevolueerd!) tot priemL = (2: 3: ...). (In dit geval is de definierende expressie voor priemL ook uitgedijd, en kost dus meer opslagruimte.)

ad r.41 Zouden we na deze evaluatie nogmaals de expressie (sub 2 priemL) evalueren, dan zijn de stappen om priemL tot de gewenste vorm te brengen niet meer nodig. Zo'n tweede evaluatie kost dan 8 stappen minder, (nml. die van regels 3,6,18,20,24,26,28 en 30).

We geven nu een tweede voorbeeld. Hierin blijkt dat bij lazy evaluation constante berekeningen niet uit een funtieromp geëlimineerd hoeven worden; althans, dat is niet nodig om herhaalde uitvoering van die berekeningen te voorkomen. Als functiedefinitie kiezen we

```
f n = (4 + 5) + n
```

en de te evalueren expressie is (f 2) + (f 3). De evaluatie is geschetst in Figuur 2; we geven nu enige toelichting bij die figuur.



FIGUUR 2. SHARING MET EEN ONDERDEEL VAN EEN FUNCTIE-DEFINITIE

Toelichting bij figuur 2.

a. De vervanging van (f 2) volgens de definitie f n = ((4+5)+n) zou als nieuwe expressie ((4+5)+2) moeten geven. Maar het onderdeel (4+5) wordt niet gedupliceerd: noemen we dit onderdeel in de funktieromp α dan kunnen we de nieuwe expressie noteren met (α+2).

b. Op regel 6 vindt de evaluatie van α plaats; deze is vereist alvorens (α+2) vervangen kan worden. De evaluatie van α heeft tot gevolg dat de funktieromp vereenvoudigd wordt!.

c. De evaluatie van (α+3) tot 12 (in regel 13) kost nu maar één stap omdat α in de funktieromp al een getalvorm heeft.

De in het begin gegeven restricties (i) en (ii) die kenmerkend zijn voor lazy evaluation, geven slechts heel impliciet de volgorde aan van de uit te voeren vervangingsstappen. We geven nu nog een alternatieve beschrijving van het evaluatieproces waarbij vooral die volgorde expliciet gemaakt wordt.

Evalueer een expressie als volgt.

(+) evalueer (expr 1 + expr 2) door:
    1. evalueer exp1 totdat een getal is bereikt;
    2. evalueer expr2 totdat een getal is bereikt;
    3. vervang de expressie door de som van die getallen.

(=) evalueer (expr1 = expr2) door:
    1. evalueer expr1 totdat een getal is bereikt;
    2. evalueer expr2 totdat een getal is bereikt;
    3. vervang de expressie door true of false al naar gelang die getallen gelijk zijn of verschillen.

(if) evalueer (if expr1 then expr2 else expr3) door:

1. evalueer expr1 totdat true of false is bereikt;
2. vervang de expressie door expr2 resp. expr3;
3. evalueer de expressie nu verder tot de gewenste vorm.

({}) evaluatie van {expr1| x <-expr2; expr3} is nogal gecompliceerd, eenvoudigheidshalve beschouwen we een voorbeeld.
evalueer {x <- expr1; 13 ÷ x} door:

1. evalueer expr1 totdat nil of de vorm (e:e') is bereikt;

Stel nil is bereikt, dan

2. vervang de expressie door nil.

Stel (e:e') is bereikt; dan

2. vervang de expressie door if 13 ÷ x then x: $^{\alpha}${x<-e'; 13 ÷ x} else α ;
3. evalueer de expressie verder tot de gewenste vorm.

(sub) evalueer (sub expr1 expr2) door:

1. evalueer expr2 totdat de vorm (e:e') is bereikt;
2. vervang de expressie door
   if $^{\alpha}$expr1 = 1 then e else sub (α-1) e';
3. evalueer de expressie verder tot de gewenste vorm.

(app) evalueer (expr1 expr2) door:

1. evalueer expr1 totdat een identifier met expliciete funktie-definitie is bereikt;
2. pas nu de regel voor die identifier toe, bijv. de vorige regel (sub).

En analoog voor de overige expressievormen. Merk op dat de recursieve structuur van dit evaluatieproces ook al uit de gegeven voorbeelden blijkt; zie nogmaals Figuur 1 en 2.

Hiermee is lazy evaluation voldoende nauwkeurig uitgelegd opdat een programmeur zich een beeld kan vormen van de efficientie (= aantal evaluatiestappen en benodigde opslagruimte) van zijn programma's.

***

We geven nu nog enige details die komen kijken by een algoritmische realisering (implementatie) van lazy evaluation.

a. Representeer expressies door voor iedere subexpressie een "cel" van de opslagruimte te nemen en daarin een aanduiding voor de expressievorm op te bergen alsmede de adressen van de cellen die de onderdelen representeren. De deling (sharing) van expressies en de vervanging van een expressie door een andere is dan eenvoudig te implementeren.

b. Houd een stapel bij met daarop verwijzingen naar de expressies (dus adressen van de cellen voor de expressie) waarvan de evaluatie al wel begonnen maar nog niet voltooid is; de meest recente op de top van de stapel. Het is dan eenvoudig om die subexpressie te vinden die aan de beurt is om geevalueerd te worden. Immers, het

evaluatieproces "bezoekt" de subexpressies in een LIFO (Last In First Out) volgorde, zoals ook al in Figuur 1 en 2 blijkt.
c. Voor iedere funktie-identifier (zoals sub) moet tijdens de compilatie-fase een programma in machine-code gegenereerd worden die betreffende vervangingsregel (zoals regel (sub)) realiseert. (Dit is allerminst triviaal; op de technieken hiervoor gaan we in dit verhaal niet in. De vertaling van expressies naar zogenaamde combinatoren is een van de mogelijkheden.) De stukjes machine-code voor regels zoals (+), (-), (if) etc zijn betrekkelijk eenvoudig.

Het evaluatieproces op zich is nu een recursief algoritme dat niet moeilijk in Pascal of zo is uit te drukken. Een dergelijke implementatie van het evaluatieproces noemen we een "interpretatie van de expressies" omdat expressies bijna letterlijk gerepresenteerd worden en die representaties per vervangingsstap bewerkt worden. Deze implementatie kan nog versneld worden (met een factor van 8 à 15) als volgt. Neem als representatie van een subexpressie nu niet een cel met als inhoud een aanduiding voor de expressievorm (+, -, sub-applicatie,...) alsmede verwijzingen naar de onderdelen, maar een cel met als inhoud de machine-code die correspondeert met de betreffende vervangingsregel alsmede verwijzingen naar de onderdelen. In plaats van de representatie van de expressie te bewerken kun je dan volstaan met die representatie te executeren! We kunnen nu spreken over "de executie van expressies". (Merk op dat de executie van deze code tot gevolg heeft dat de code herhaaldelijk wordt gewijzigd en overschreven (corresponderende met de vervanging van subexpressies door andere). Het opstellen van zichzelf wijzigende code geeft nogal eens foute programma's en wordt in het algemeen ten sterkste afgeraden. Maar hier zijn de wijzigingen zonder gevaar: per definitie is de code-transformatie korrekt.)