Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

J. Heering

Programmageneratoren

Informatica/Afdeling Programmatuur Notitie CS-N8607 Augustus

Programming generators

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

Programmageneratoren

Jan Heering
Centrum voor Wiskunde en Informatica,
Kruislaan 413, 1098 SJ Amsterdam.

Het snel in omvang toenemende gebruik van programmageneratoren en de bijbehorende specificatietalen ("vierde-generatie talen") kan licht de indruk doen ontstaan dat het hier om een nieuwe en revolutionaire ontwikkeling gaat. In feite is het echter de natuurlijke en geleidelijke voortzetting van een lijn van ontwikkeling die vrijwel even oud is als de computer zelf. Het terrein is intussen zeer omvangrijk. Terwijl tal van huis-tuin-en-keukengeneratoren hun nut dagelijks in de praktijk bewijzen, wordt er in de software-laboratoria gestaag gewerkt aan meer esoterische mogelijkheden als het genereren van programma's op basis van voorbeelden en het automatisch afleiden van efficiënte gespecialiseerde versies uit algemene programma's met behulp van partiële evaluatie.

1986 CR Categories: D.1.2 [**Programming Techniques**]: Automatic Programming; I.2.2 [**Artificial Intelligence**]: Automatic Programming; I.2.6 [**Artificial Intelligence**]: Learning - *induction.*

1980 Mathematics Subject Classification: 68B99 [**Software**].

Trefwoorden: programmagenerator, specificatietaal, vierde-generatie taal, genereren van programma's op basis van voorbeelden, inductie, editing by example, partiële evaluatie.

I think the first step to tell us that we could actually use a computer to write programs was Betty Holberton's "Sort-Merge Generator." You fed it the specifications of the files you were operating on, and the Sort-Merge Generator produced the program to do the sorting and merging . . . Of course, at that time [1952] the Establishment promptly told us—at least they told me quite frequently—that a computer could not write a program; it was totally impossible; that all that computers could do was arithmetic, and that it couldn't write programs; that it had none of the imagination and dexterity of a human being. I kept trying to explain that we were wrapping up the human being's dexterity in the program that he wrote, the generator, and that of course we could make a computer do these things so long as they were completely defined.

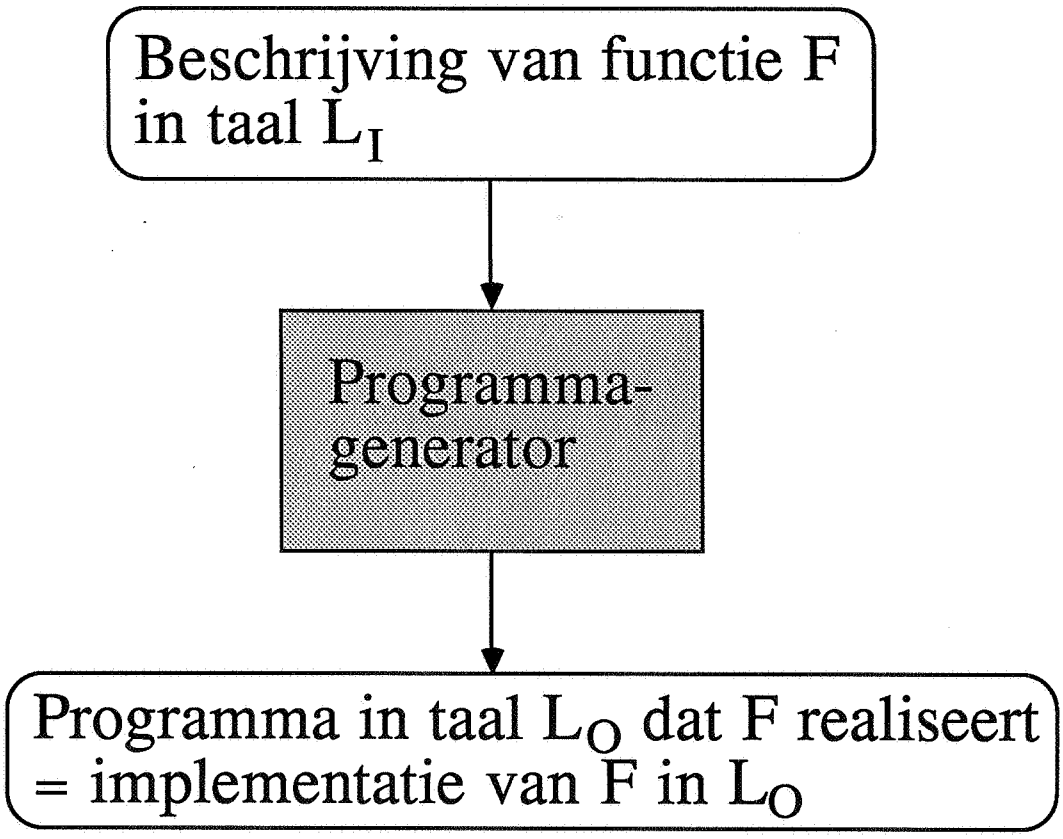
Grace Murray Hopper, ACM History of Programming Languages Conference, juni 1978. R.L. Wexelblat (ed.), *History of Programming Languages*, Academic Press, 1981, p. 9.

Notitie CS-N8607
Centrum voor Wiskunde en Informatica
Postbus 4079, 1009 AB Amsterdam

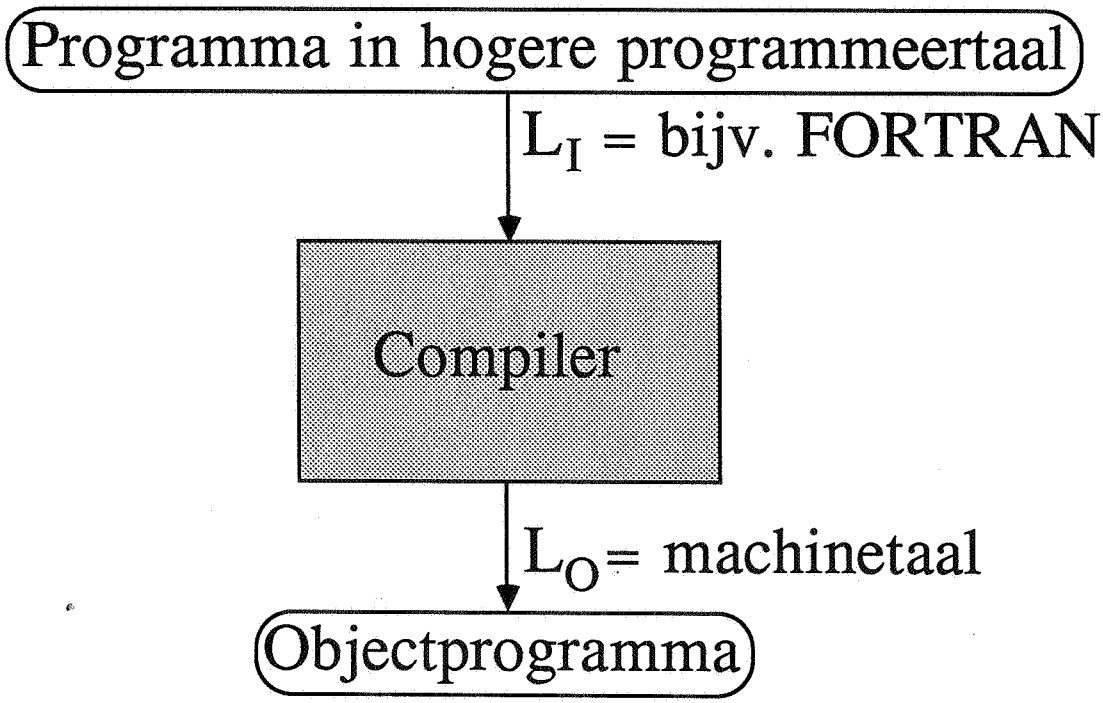
INHOUD

1. Algemeen
2. Editing by example
3. Partiële evaluatie
4. Literatuur

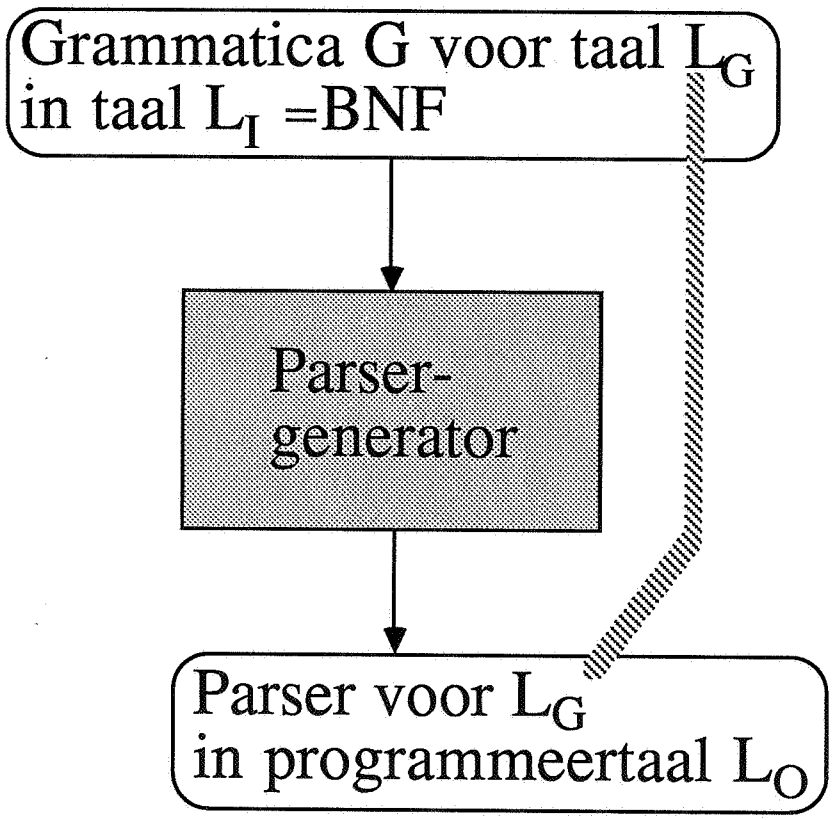
1. ALGEMEEN



Bekendste voorbeeld:



Parsergenerator



- Grammatica G brengt zinnen voort die samen taal L_G vormen

- Gegenereerde parser beslist of gegeven zin tot L_G behoort en levert eventueel afleidingsboom

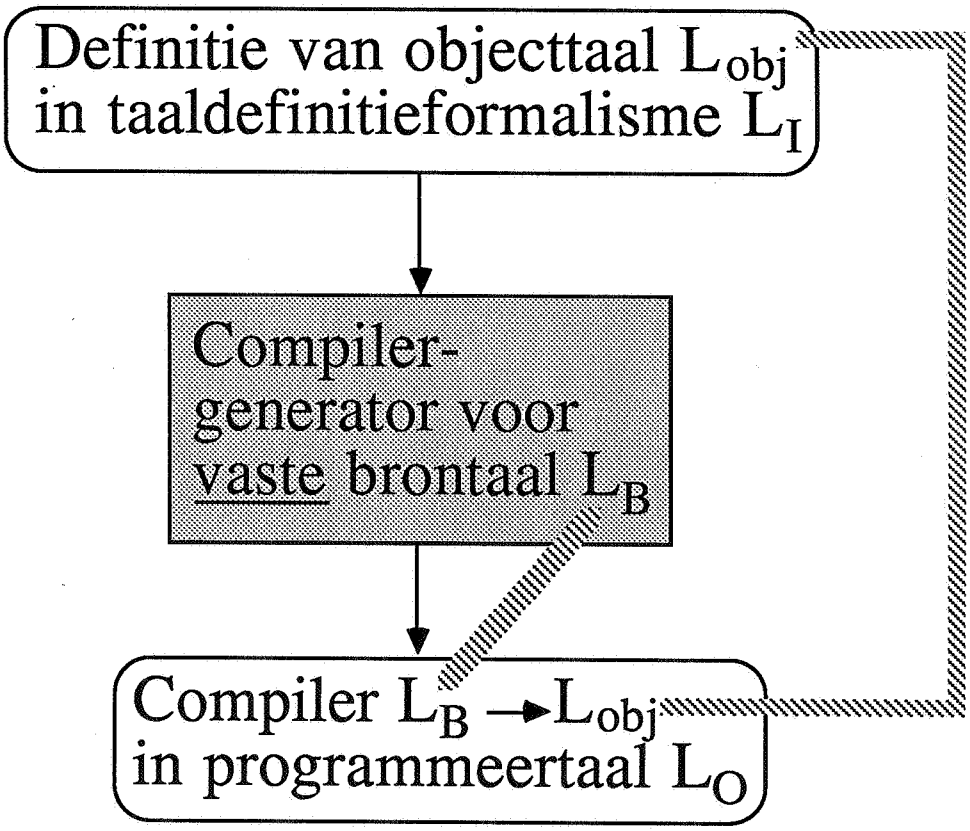
- Meeste parsergenerators zoals YACC accepteren geen willekeurige contextvrije grammatica's, maar alleen LR(1)-, LL(1)- of LALR(1)-grammatica's [JOH79, AU72]

- (Verstandige) beperking van de klasse van geaccepteerde grammatica's maakt het mogelijk parsers te genereren met
  - gegarandeerd goede efficiëntie
  - gegarandeerd goede foutenafhandeling (zoals bijv. in PGEN [FR81])

- Zie [TOM86] voor parsergenerator die willekeurige contextvrije grammatica's accepteert

- Open problemen:
  - Genereren van incrementele parsers
  - Genereren van goede foutenafhandeling in het algemene geval
  - Automatisch afleiden van goede parser voor specifieke grammatica uit universele parser (zie ook hfdst. 3)

Compilergenerator (1)



- De gegenereerde compiler accepteert programma's in L_B en vertaalt ze naar L_{obj}

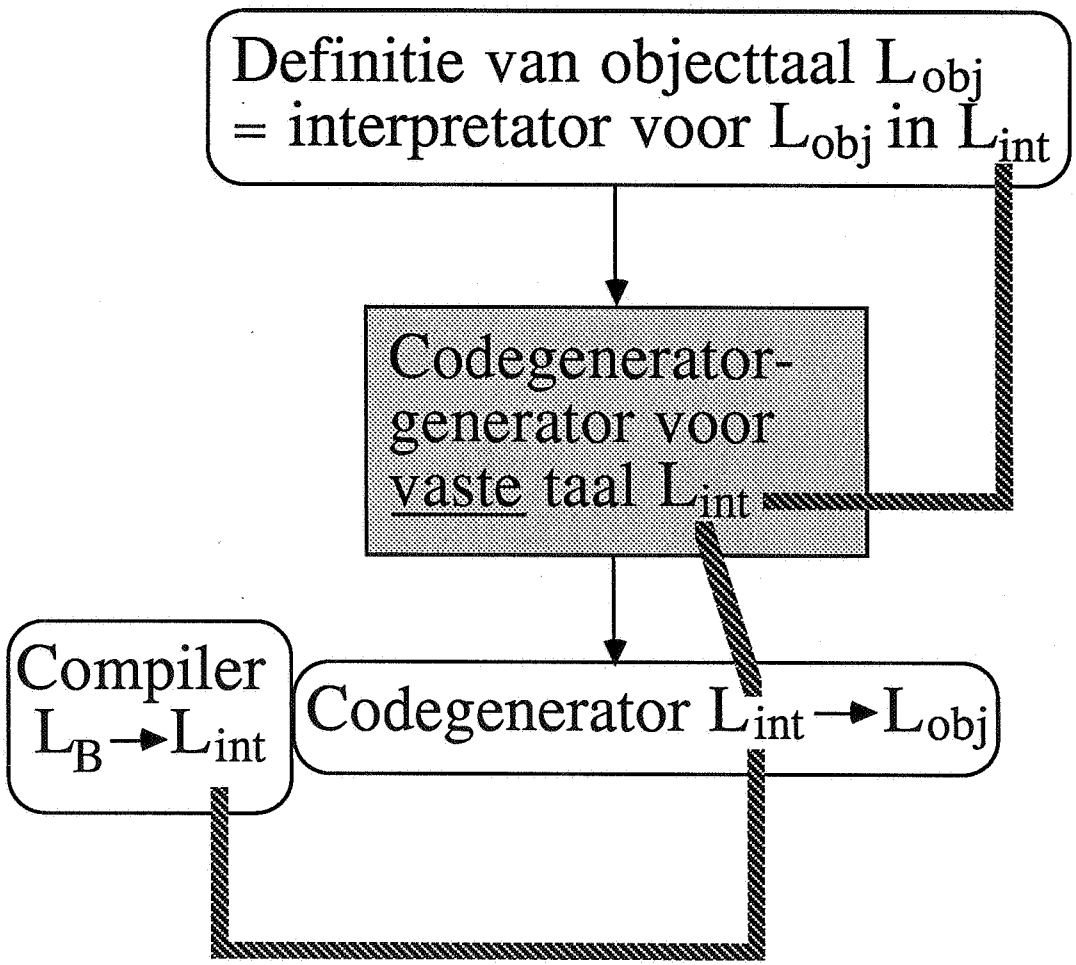
Bijv.

- Brontaal L_B = FORTRAN

- Generator levert op basis van definities (in L_I) van verschillende machinetalen FORTRAN-compilers voor die machines

- Maken van definitie van gegeven machinetaal L_{obj} veel minder werk dan schrijven van compiler FORTRAN$\rightarrow L_{obj}$ dus generator neemt meeste werk over

Compilergenerator (1) (vervolg)



- L_{int}= laag-niveau tussentaal

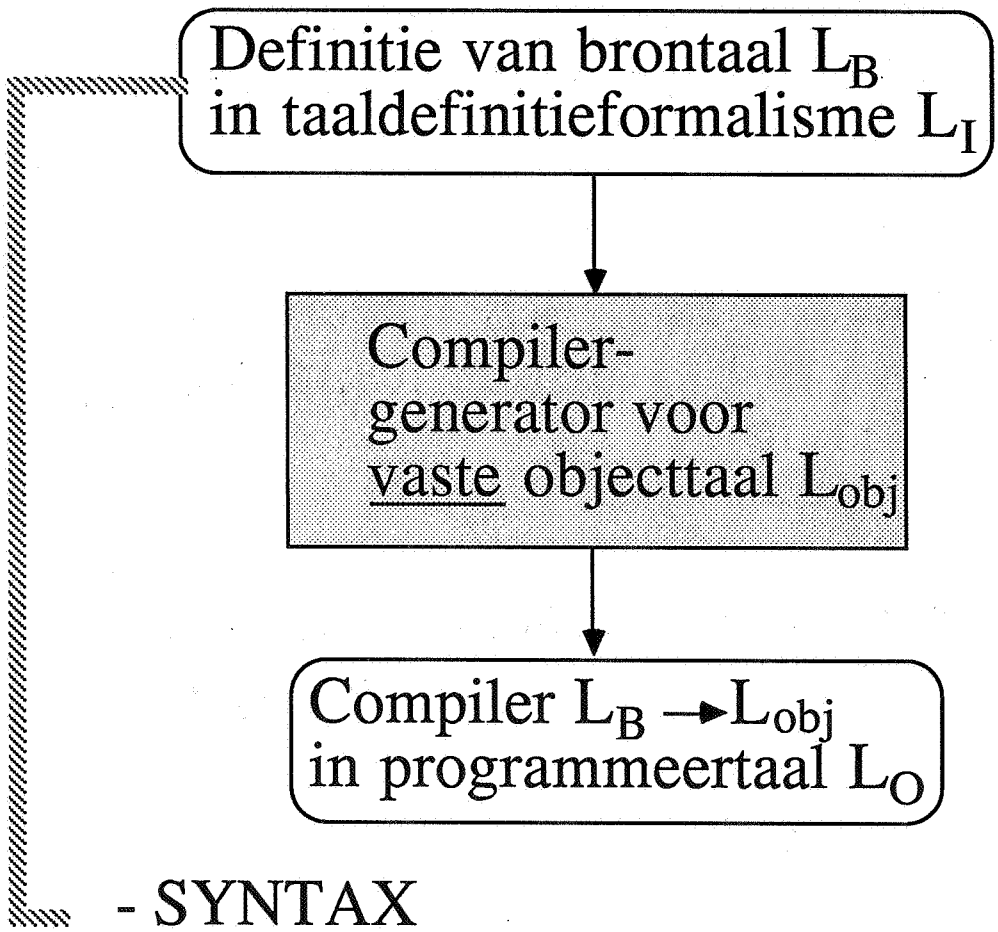
- Interpretatieve definitie van L_{obj} geeft betekenis van L_{obj}-instructies in termen van L_{int}

- Codegenerator $L_{int} \rightarrow L_{obj}$ heeft juist betekenis van L_{int}-instructies in termen van L_{obj} nodig!

- De codegenerator-generator moet dus o.a. de definitie van L_{obj} inverteren

- Zie ook [CAT80]

Compilergenerator (2)



- SYNTAX
- ABSTRACTE SYNTAX
- SEMANTIEK

zie [PAU82] en hfdst. 3

☐ Brontalen L_I voor programmagenerators worden vaak gekwalificeerd als

- *non-procedural*
L_I-specificaties bevatten veelal weinig details over *hoe* gespecifeerde taak verricht moet worden — denk aan BNF-grammatica als input voor parsergenerator

- *very high-level*
natuurlijke voortzetting van ontwikkeling die destijds leidde tot invoering van hogere programmeertalen, maar. . .
de very high-level language van vandaag is de low-level language van morgen

- *fourth generation languages*
waarbij hogere programmeertalen, assemblertalen en machinetalen resp. de derde, tweede en eerste generatie zouden vormen

☐ Brontalen L_I bestrijken veelal *beperkt applicatiegebied*

☐ Gebruik programmagenerators

- vermijdt keer op keer uitvinden van het wiel
- werkt *productiviteitsverhogend*
- werkt *uniformiserend*
  bijv. alle door parsergenerator geproduceerde parsers geven zelfde type foutmeldingen
- maakt experimenteren en modificeren gemakkelijker
- is ook interessant als er geen programma's van productiekwaliteit opgeleverd worden, d.w.z. voor *prototyping*

□ De programmeertalen Lisp [WH81] en Prolog [CM84] lenen zich bij uitstek voor programmagenerators

- Lisp-programma = Lisp-lijst = in Lisp manipuleerbaar object
  Prolog-clause = Prolog-term = in Prolog manipuleerbaar object

- Generator is in dit geval Lisp- resp. Prolog-programma dat Lisp- resp. Prolog-programma produceert

☐ Verdere voorbeelden van programmagenerators

- "Application generators" — zie [HKN85]
- Generators die hardware-simulators produceren op basis van architectuurbeschrijvingen, bijv. MISS [OOS82]
- Generators die syntaxgestuurde editors produceren op basis van (verrijkte) BNF-grammatica's, bijv. GANDALF [GAN85], Cornell Synthesizer Generator en MENTOR [KLI83]
- Generators die programmeeromgevingen produceren op basis van formele taaldefinities, bijv. PSG [SNE85]

2. EDITING BY EXAMPLE

☐ Leerzame oefening: zet met de *vi* screen editor onder UNIX regels als

```
SPLINE FROM 100,50%
TO 160,70
...
```

door hele tekst om naar

```
SPLINE FROM 100,50 TO 160,70 ...
```

d.w.z. als regel eindigt op % verwijder dan % en plak volgende regel erachter

(1) Eerste instantie gaat met commando's

`/%$` zoek eerstvolgende % aan einde regel
`xJ` verwijder % en "join" volgende regel

(2) Elke volgende instantie gaat met commando

`nxJ`

☐ Hoe laat je in een keer alle wijzigingen uitvoeren?

(1) Zet (recursieve) macro `nxJ@Z` in *vi*-register `Z`

(2) Geef commando's

`/%$` als boven

`@Z` voer macro in `Z`-register uit

☐ Te moeilijk — zo wil je het in ieder geval niet doen, maar hoe dan wel?

(1) Voer enkele veranderingen bij wijze van voorbeeld uit

```
SPLINE FROM 100,150%
TO 150,75
→ SPLINE FROM 100,150 TO 150,75

TO 150,75%
TO 160,70
→ TO 150,75 TO 160,70
```

(2) Laat editor gegeven voorbeelden generaliseren tot editprogramma dat wijziging door hele tekst uitvoert

☐ Voorbeelden generaliseren = *inductie* [HOL86]

☐ Inductie heeft *probabilistisch* karakter

☐ Voorbeelden te zien als *specificatie van zeer hoog niveau* van het te genereren programma

☐ Op dit moment alleen interessant voor *beperkte* klassen van programma's
Genereren van willekeurige programma's op basis van voorbeelden te moeilijk

☐ Algemeen principe

(*a*) Klasse van programma's K

(*b*) Complexiteitsmaat μ op K

(*c*) Stel voorbeelden $V = \{a_i \rightarrow b_i\}_{i=1}^{m}$

Zoek programma P in K zodanig dat

(1) $P(a_i) = b_i \quad (i = 1, \ldots, m)$,
d.w.z. *P correct op alle voorbeelden*

(2) $\mu(P)$ minimaal,
d.w.z. *P zo eenvoudig mogelijk*

☐ Bij beperkte K hoeft er niet voor elke verzameling voorbeelden V een programma P in K te zijn dat aan eis (1) voldoet!

☐ Eis (2) is *Ockham's scheermes* relatief t.o.v. de complexiteitsmaat μ

☐ Editing by example à la Nix [NIX85]

Klasse van programma's K bevat alleen *stringherschrijfregels* ("gap programs")

patroon $p \rightarrow$ substitutie s

met

$p = c_0 X_1 c_1 \cdots c_{n-1} X_n c_n \ (n \geqslant 0)$

c_j: strings van symbolen,
niet leeg voor $j \geqslant 1$

X_j: variabelen, $X_j \neq X_k$ als $j \neq k$,
d.w.z. p *lineair*

$s =$ string van symbolen en variabelen X_j

Bijv.

R: praline $\rightarrow$ bonbon
($n=0$, $c_0=$praline)
R(praline)=bonbon

Q: Xke $\rightarrow$ $X X$o
($n=1$, c_0 leeg, $c_1=$ke)
Q(cake)=cacao ($X=$ca)

P: XvYaat $\rightarrow$ avYXo
P(advocaat)=avocado

Steeds *meest linkse* match nemen, bijv.

S: Xee$Y$r $\rightarrow$ Xark
S(kweepeer)=kwark ($X=$kw)

maar *niet*

S(kweepeer)=kweepark
($X=$kweep)

d.w.z. na matching van p mag c_j geen substring van de waarde van X_j zijn

☐ Nu complexiteitsmaat μ op string-herschrijfregels definiëren

☐ μ primair op patronen nodig

$p = c_0 X_1 c_1 \cdots X_n c_n \ (n \geqslant 0)$

$l = \text{lengte}(c_0 c_1 \cdots c_n)$

= lengte van het "skelet" van p

$l \geqslant n$, want c_j niet leeg voor $j \geqslant 1$

$l \geqslant 1$, want c_0 niet leeg als $n = 0$

$$\mu(p) = \frac{1}{l^2 - n + 1}$$

$\mu(p)$ minimaal als

(1) l maximaal,

d.w.z. skelet van p zo lang mogelijk

(2) n minimaal,

d.w.z. zo min mogelijk variabelen in p

Bijv.

$$\mu(X\mathtt{a}) = 1 \quad (l = 1,\ n = 1)$$

$$\mu(Y\mathtt{cb}X\mathtt{a}) = \frac{1}{8} \quad (l = 3,\ n = 2)$$

□ Genereer nu herschrijfregel $p \rightarrow s$ op basis van gegeven verzameling voorbeelden $V = \{a_i \rightarrow b_i\}_{i=1}^{m}$ (a_i, b_i strings) in 2 stappen:

(I) Zoek patroon p dat

- a_i matcht voor alle $i = 1, \ldots, m$,
  d.w.z. als $p = c_0 X_1 c_1 \cdots X_n c_n$ dan

$$a_1 = c_0 \sigma_{1,1} c_1 \cdots \sigma_{1,n} c_n$$
$$\cdots$$
$$a_m = c_0 \sigma_{m,1} c_1 \cdots \sigma_{m,n} c_n$$

  voor zekere strings $\sigma_{i,j}$
- minimale $\mu(p)$ heeft

(II) Zoek bijpassende substitutie s zodat

$$(p \rightarrow s)(a_i) = b_i \quad (i = 1, \ldots, m)$$

□ Als (I) en (II) slagen, pas dan gevonden regel door hele tekst toe

□ Goede *undo* nodig om effecten van onbedoelde generalisaties ongedaan te maken!

☐ (I) faalt: er is geen stringherschrijfregel die correct is op alle voorbeelden

☐ (II) faalt: er is geen stringherschrijf-regel $p \rightarrow s$ met p uit stap (I) die correct is op alle voorbeelden
Mogelijk voldoet een $p' \rightarrow s$ met $p' \neq p$, maar de procedure vindt p' niet (geen backtracking)

☐ Stap (I) valt uiteen in

(I*a*) Zoek de *langste gemeenschappelijke deelrij* C van $a_1 \cdots a_m$
C wordt het skelet van p

(I*b*) Voeg in C (zo min mogelijk) variabelen in zodat resulterende p alle a_i's matcht

☐ *Heuristische* procedures voor (I*a*) en (I*b*) nodig om in acceptabele tijd resultaat te krijgen

☐ Slechts 1 voorbeeld ($m=1$) :
$C=a_1$, $p=a_1$, dus geen generalizatie

☐ Bijv. ($m=2$):

$$V=\{\mathsf{abubav} \rightarrow \mathsf{ababa},\ \mathsf{cdduddcv} \rightarrow \mathsf{addcddc}\}$$

Stap (I*a*) geeft:

$$C=\mathsf{uv}$$

Stap (I*b*) geeft:

$$p=X_1\mathsf{u}X_2\mathsf{v}$$

$$\sigma_{1,1}=\mathsf{ab} \quad \sigma_{1,2}=\mathsf{ba}$$
$$\sigma_{2,1}=\mathsf{cdd} \quad \sigma_{2,2}=\mathsf{ddc}$$

Nu stap (II). . .

. . . bijpassende s zoeken

$$
\begin{aligned}
b_1 &= \\
&= \mathsf{ababa} \\
&= \sigma_{1,1}\mathsf{aba} \\
&= \sigma_{1,1}\sigma_{1,1}\mathsf{a} \\
&= \sigma_{1,1}\mathsf{a}\sigma_{1,2} \\
&= \mathsf{a}\sigma_{1,2}\mathsf{ba} \\
&= \mathsf{a}\sigma_{1,2}\sigma_{1,2} \quad \text{☞} \\
&= \mathsf{ab}\sigma_{1,1}\mathsf{a} \\
&= \mathsf{aba}\sigma_{1,2}
\end{aligned}
$$

$$
\begin{aligned}
b_2 &= \\
&= \mathsf{addcddc} \\
&= \mathsf{a}\sigma_{2,2}\mathsf{ddc} \\
&= \mathsf{a}\sigma_{2,2}\sigma_{2,2} \quad \text{☞} \\
&= \mathsf{addc}\sigma_{2,2} \\
&= \mathsf{add}\sigma_{2,1}\mathsf{c}
\end{aligned}
$$

$$s = \mathsf{a}X_2X_2$$

☐ Meestal levert deze procedure op basis van 2 of 3 voorbeelden de gewenste stringherschrijfregel *indien er zo'n regel is*

☐ Vergroting van klasse K is wenselijk, maar maakt inductieprobleem veel moeilijker

☐ In stap (I*a*) opgeleverde skelet C bevat vaak "ruis" ("toevallige" gemeenschappelijke symbolen in de a_i's)
Deze kan tot op zekere hoogte (heuristisch) uitgefilterd worden door naar groepjes symbolen te kijken i.p.v. naar enkele symbolen

☐ Editing by example is beschikbaar in the *U* editor van Yale University

3. PARTIELE EVALUATIE

☐ Partiële evaluatie = evaluatie van programma's (procedures) waarvan niet alle argumenten volledig bekend zijn = vorm van *symbolische evaluatie*

☐ Bijv. stel dat van programma $P = P(x,y)$

- argument x bekende waarde a heeft
- argument y nog onbekend is

"Zo veel mogelijk" evalueren in P op basis van bekende waarde a geeft gespecialiseerd *residuprogramma* $P_a = P_a(y)$ met

$$P_a(b) = P(a,b)$$

Een gewone evaluator accepteert $P(a,y)$ niet!

☐ P_a heeft nog maar 1 argument, terwijl P er 2 had

☐ Als het goed is, gaat berekening van $P_a(b)$ efficiënter dan van $P(a,b)$, omdat er tijdens partiële evaluatie al het een en ander gedaan is

☐ Notie "zo veel mogelijk evalueren" met korreltje zout nemen, want in het algemeen is voor gegeven $P = P(x,y)$ en a residu P_a niet eenduidig te bepalen (zie vervolg)

☐ (I) Eenvoudig voorbeeld [ERS82]

$F(n,x)$
begin
  $y:=1$
  while $n>0$
  do
    while n even
    do
      $n:=\frac{n}{2}$
      $x:=x^2$
    od
    $n:=n-1$
    $y:=y.x$
  od
end

$F(n,x)$ berekent de functie x^n

Gespecialiseerde versie $F_5(x)$ die x^5 berekent, uit $F(5,x)$ afleiden met partiële evaluatie. . .

. . . $n=5$, x ongebonden:

$y:=1$
[$5>0$]
  [5 niet even]
  $n:=4$
  $y:=1.x$ $\triangleright$
[$4>0$]
  [4 even]
    $n:=2$
    $x:=x^2$ $\triangleright$
  [2 even]
    $n:=1$
    $x:=x^2$ $\triangleright$
  [1 niet even]
  $n:=0$
  $y:=y.x$ $\triangleright$
[0 niet >0]

Residuprogramma

$F_5(x)$
begin
$y := 1.x$
$x := x^2$
$x := x^2$
$y := y.x$
end

□ F_5 bevat helemaal geen tests meer (is lineair), maar in het algemeen is dat niet te verwachten

□ F_5 kan nog verder vereenvoudigd worden met $1.x = x$, etc.

□ (II) Moeilijker geval

$R(x)=$
if $x = nil$
then *nil*
else
if $head(x) = \mathsf{a}$
then $\mathsf{b}.R(tail(x))$
else $head(x).R(tail(x))$
fi
fi

R vervangt alle **a**'s door **b**'s in string x
nil is lege string
. is stringconcatenatie-operator
$head(x)$ geeft eerste symbool van x
$tail(x)$ geeft alles behalve eerste symbool

□ Stel x is een string beginnend met een **a**, d.w.z. $x = \mathsf{a}.y$
Partiële evaluatie van $R(\mathsf{a}.y)$. . .

if a.$y = nil$
then nil
else
  if $head$(a.y) = a
  then b.R($tail$(a.y))
  else $head$(a.y).R($tail$(a.y))
  fi
fi

reduceert met a.$y \neq nil$ tot

if $head$(a.y) = a
then b.R($tail$(a.y))
else $head$(a.y).R($tail$(a.y))
fi

reduceert met $head$(a.y) = a tot

b.R($tail$(a.y))

reduceert met $tail$(a.y) = y tot

b.R(y)

□ Als $x = y.\mathsf{a}$ verwacht je natuurlijk op dezelfde manier

$$R(y.\mathsf{a}) \rightarrow R(y).\mathsf{b}$$

Voor alle zekerheid even nagaan:

if $y.\mathsf{a} = nil$
then nil
else
  if $head(y.\mathsf{a}) = \mathsf{a}$
  then $\mathsf{b}.R(tail(y.\mathsf{a}))$
  else $head(y.\mathsf{a}).R(tail(y.\mathsf{a}))$
  fi
fi

reduceert met $y.\mathsf{a} \neq nil$ tot

if $head(y.\mathsf{a}) = \mathsf{a}$
then $\mathsf{b}.R(tail(y.\mathsf{a}))$
else $head(y.\mathsf{a}).R(tail(y.\mathsf{a}))$
fi

Wat nu? Er zijn blijkbaar 3 gevallen

$head(y.\mathsf{a}) = \mathsf{a}$ als $y = nil$ of $y = \mathsf{a}.u$
$head(y.\mathsf{a}) \neq \mathsf{a}$ als $y = q.u$ $(q \neq \mathsf{a})$

$y = nil$:

$\mathbf{b}.R(tail(nil.\mathbf{a})) \rightarrow$
$\mathbf{b}.R(tail(\mathbf{a})) \rightarrow$
$\mathbf{b}.R(nil) \rightarrow$
$\mathbf{b}.nil \rightarrow$
$\mathbf{b}$

$y = \mathbf{a}.u$:

$\mathbf{b}.R(tail(\mathbf{a}.u.\mathbf{a})) \rightarrow$
$\mathbf{b}.R(u.\mathbf{a})$

$y = q.u \quad (q \neq \mathbf{a})$:

$head(q.u.\mathbf{a}).R(tail(q.u.\mathbf{a})) \rightarrow$
$q.R(tail(q.u.\mathbf{a})) \rightarrow$
$q.R(u.\mathbf{a})$

□ Laatste 2 gevallen zijn in principe nog hetzelfde als oorspronkelijke geval (met u i.p.v. y)
De laatste **a** is "onbereikbaar" en wordt op deze manier nooit een **b**, d.w.z. het resultaat $R(y).\mathbf{b}$ komt er niet uit!

☐ In het voorgaande is aangenomen dat de partiële evaluator o.a. de volgende eigenschappen van strings kent:

$$q.y \neq nil$$
$$head(q.y) = q$$
$$tail(q.y) = y$$

Een gewone evaluator heeft deze eigenschappen nooit nodig

☐ $R(y.\mathbf{a}) \rightarrow R(y).\mathbf{b}$ volgt direct uit de distributieve wet

$$R(x.y) = R(x).R(y)$$

maar daarover beschikt de partiële evaluator niet en het is moeilijk hem automatisch op basis van de definitie van R te vinden

☐ Er zijn nog meer van dergelijke wetten, bijv.

$$R(R(x)) = R(x)$$
$$R(tail(x)) = tail(R(x))$$

Zie ook voorbeeld (IV)

☐ (III) Zeer moeilijk geval

Gewone parser ontleedt string volgens *vaste* grammatica

Universele parser werkt voor *willekeurige* grammatica

$$P(g,s)$$

g: contextvrije grammatica in BNF
s: volgens g te ontleden string

Bijv. P = *Earley-parser* [AU72, §4.2.2]

☐ Prijs: universele parser P in het algemeen inefficiënt voor specifieke grammatica's

☐ Kun je voor specifieke G goede parser P_G met partiële evaluatie afleiden uit universele P?

Zie [DYB85] voor poging om (afgezwakte versie van) de Earley-parser partieel te evalueren voor specifieke grammatica's

☐ Partiële evaluatie is

- een belangrijk algemeen principe dat inzicht geeft in
  - programmageneratie
  - de relatie tussen compilatie en interpretatie (zie vervolg)
  - de noties *compile-time* en *run-time*
  - type checking
- een techniek die soms *met de hand* toegepast kan worden om efficiënte gespecialiseerde versies af te leiden uit algemene programma's
- een techniek die (vooralsnog op zeer beperkte schaal) gebruikt kan worden voor *automatische* specialisatie van programma's

☐ Conventionele (zg. *imperatieve*) programmeertalen zijn niet erg geschikt voor partiële evaluatie

Je kunt *niet lokaal*, d.w.z. *zuiver syntactisch*, vaststellen of je een statement ergens in het programma kunt executeren zonder dat je eerst alle (dynamisch) voorafgaande statements hebt uitgevoerd (maar dat wil je juist niet!)

Oorzaken

- Globale variabelen
- Meerdere assignments aan dezelfde variabele
- Aliasing, gebruik van pointers, etc.

☐ Beter geschikt zijn programmeertalen die door logica en wiskunde zijn geïnspireerd:

- *Functionele talen* zoals Lispkit Lisp, FP, ML, . . .
  Zie [HEN80] en voorbeeld (II)
- *Termherschrijfregels*
  Zie voorbeeld (IV)
- *Logische talen* zoals
  (concurrent) Prolog, . . .

☐ Doel: je wilt programma's kunnen opvatten en manipuleren als wiskundige formules — zie ook [MEE86]

□ De relatie tusen interpretatie en compilatie vanuit het gezichtspunt van partiële evaluatie [ERS82]

(A) Compilatie

- Interpretator voor brontaal L_B

  $Int = Int(p,x)$

  p: L_B-programma met 1 argument
  x: argument van p

Voor specifieke P en a geldt dus

$Int(P,a) = P(a)$

Int zelf geschreven in L_{Int}

- Partiële evaluator voor L_{Int}

  $PE = PE(q,u)$

  q: L_{Int}-programma met 2 argumenten
  u: eerste argument van q

Voor specifieke $Q = Q(x,y)$ en b geldt dus

$PE(Q,b) = Q_b(y)$

□ Pas *PE* toe op *Int* voor specifiek L_B-programma $P=P(x)$ met ongebonden argument x, d.w.z. neem $q=Int$ en $u=P$:

$$PE(Int,P)=Int_P(x)$$

Int_P is L_{Int}-programma met

$$Int_P(a)=Int(P,a)=P(a)$$

Int_P is

- op P toegespitste *Int*
- *vertaling* van P in L_{Int}
- *objectcode* voor P

□ *Int* heeft i.h.a. globaal de volgende structuur

```
Int(p,x)
begin
   while stmnt:=nextstmnt(p)
   do
      case stmnt
         <if-stmnt>: «actie»   ▷
         <proccall-stmnt>: «actie»   ▷
         <assigment-stmnt>: «actie»   ▷
            . . .
      esac
   od
end
```

Bij uitvoering van $PE(Int,P)$ wordt voor elk statement van P de bijbehorende (gedeeltelijk geëvalueerde) «actie» van *Int* aan het residu Int_P toegevoegd

(B) Compilergeneratie

□ *PE* moet nu een L_{Int}-*autoprojector* zijn, d.w.z. een partiële evaluator voor L_{Int}-programma's *die zelf ook in* L_{Int} *geschreven is en dus op zichzelf toegepast kan worden*

□ Neem $q = PE$ en $u = Int$

$$PE(PE, Int) = PE_{Int}(u')$$

PE_{Int} levert bij L_B-programma P behorende objectcode Int_P:

$$PE_{Int}(P) = PE(Int, P) = Int_P$$

PE_{Int} is

- op *Int* toegespitste *PE*
- *compiler* $L_B \rightarrow L_{Int}$

L_{Int}-autoprojector genereert dus middels zelfapplicatie compilers op basis van interpretatieve taaldefinities

☐ Wat is/wordt er gedaan?

- Partiële evaluator voor Lisp [BHOS76]
- Partiële evaluator voor Prolog [KOM82]
- Succesvolle compilergeneratie "onder laboratoriumomstandigheden" met de MIX autoprojector [JSS85, SES85]
- Pogingen om gespecialiseerde parsers af te leiden uit universele parser met de MIX autoprojector [DYB85]
- Onderzoek van partiële evaluatie in het kader van algebraïsche specificatie en termherschrijfsystemen begint op gang te komen
  Algebraïsch framework in principe zeer geschikt

□ *Volledige* partiële evaluatie =
"zoveel mogelijk berekenen op basis van onvolledige informatie" =
programma's met onvolledig bekende argumenten *in normaalvorm brengen*

(IV) Voor *zeer beperkte* klassen van programma's kan dat, bijv. voor Boolese expressies met signatuur

constanten $T, F : Bool$

functies $\neg : Bool \to Bool$ (*niet*)
$\vee : Bool \times Bool \to Bool$ (*of*)
$+ : Bool \times Bool \to Bool$ (*xof*)
$. : Bool \times Bool \to Bool$ (*en*)

variabelen $X, Y, Z, \ldots : Bool$

Voorbeeld van Boolese expressie met variabelen

$$Q(X,Y) = (X + X) \vee (Y.Y)$$
$$Q(T,Y) = (T + T) \vee (Y.Y)$$

□ Interpretator (*termherschrijfsysteem*) *IntBool* voor "*Bool*-programma's", d.w.z. Boolese expressies zonder variabelen

IntBool
begin
$\neg T \to F$
$\neg F \to T$
$T \vee T \to T$
$T \vee F \to T$
$F \vee T \to T$
$F \vee F \to F$
$T + T \to F$
$T + F \to T$
$F + T \to T$
$F + F \to F$
$T.T \to T$
$T.F \to F$
$F.T \to F$
$F.F \to F$
end

□ *IntBool* reduceert elke Boolese expressie zonder variabelen tot T of F, bijv.

$$
\begin{aligned}
&Q(F,T)= \\
&(F+F)\vee(T.T) \rightarrow \\
&F\vee(T.T) \rightarrow \\
&F\vee T \rightarrow \\
&T
\end{aligned}
$$

Reductie kan i.h.a. in vele verschillende volgordes maar levert steeds hetzelfde resultaat

□ *IntBool* werkt ook als partiële evaluator, bijv.

$$
\begin{aligned}
&Q(T,Y)=(T+T)\vee(Y.Y) \rightarrow \\
&F\vee(Y.Y)
\end{aligned}
$$

IntBool kan het residu $F\vee(Y.Y)$ niet verder reduceren en is dus onvolledig, want met extra regels gaat het wel. . .

$F \vee (Y.Y) \rightarrow$ met $F \vee X = X$
$Y.Y \rightarrow$ met $Y.Y = Y$
Y

☐ *Volledige* partiële evaluator *IntBool**

*IntBool**
begin
$\neg X \rightarrow X + T$
$X \vee Y \rightarrow (X.Y) + (X + Y)$
$X + F \rightarrow X$
$X + X \rightarrow F$
$X.F \rightarrow F$
$X.T \rightarrow X$
$X.X \rightarrow X$
$X.(Y + Z) \rightarrow (X.Y) + (X.Z)$
end

Herschrijving modulo associativiteit en commutativiteit van . en +. . .

. . . d.w.z. de associatieve en commutatieve wetten

$$(X+Y)+Z=X+(Y+Z)$$
$$X+Y=Y+X$$
$$(X.Y).Z=X.(Y.Z)$$
$$X.Y=Y.X$$

zitten niet als herschrijfregel in *IntBool** (dat geeft oneindige loops), maar moeten tijdens regelmatching worden toegepast, bijv.

$Q(X,Y)=$
$(X+X)\vee(Y.Y)\rightarrow$
$F\vee(Y.Y)\rightarrow$
$F\vee Y\rightarrow$
$(F.Y)+(F+Y)\rightarrow$ (modulo $X.Y=Y.X$)
$F+(F+Y)\rightarrow$ (modulo $X+Y=Y+X$)
$F+Y\rightarrow$ (modulo $X+Y=Y+X$)
Y

Dus *a fortiori*

$$Q(T,Y)\rightarrow Y$$

☐ Volledigheid van *IntBool** is niet vanzelfsprekend — bewijs blijft hier achterwege

☐ Boolese expressies zijn een bijzonder geval, want meestal is volledige partiële evaluatie *principieel uitgesloten* — zie [HEE86]

☐ We zullen het dus moeten doen met (zeer) onvolledige partiële evaluators!

4. REFERENTIES

[AU72] A.V. Aho & J.D. Ullman, *The Theory of Parsing, Translation, and Compiling*, Vol. I, *Parsing*, Prentice-Hall, 1972.

[BHOS76] L. Beckman, A. Haraldson, O. Oskarsson & E. Sandewall, A partial evaluator and its use as a programming tool, *Artificial Intelligence*, **7** (1976), pp. 319-357.

[CAT80] R.G.G. Cattell, Automatic derivation of code generators from machine descriptions, *ACM Transactions on Programming Languages and Systems*, **2** (1980), 2, pp. 173-190.

[CM84] W.F. Clocksin & C.S. Mellish, *Programming in Prolog*, 2nd ed., Springer-Verlag, 1984.

[DYB85] H. Dybkjær, Parsers and partial evaluation: an experiment, Rapport 85-7-15, Datalogisk Institut, Universiteit van Kopenhagen, Kopenhagen, juli 1985.

[ERS82] A.P. Ershov, Mixed computation: potential applications and problems for study, *Theoretical Computer Science*, **18** (1982), pp. 41-67.

[FR81] G. Florijn & G. Rolf, PGEN - A general purpose parser generator, Rapport IW 157/81, Centrum voor Wiskunde en Informatica, Amsterdam, 1981.

[GAN85] *The Journal of Systems and Software*, **5** (1985), 2 (nummer gewijd aan de GANDALF editorgenerator).

[HEE86] J. Heering, Partial evaluation and ω-completeness of algebraic specifications, *Theoretical Computer Science*, **43** (1986), pp. 149-167.

[HEN80] P. Henderson, *Functional Programming*, Prentice-Hall International, 1980.

[HKN85] E. Horowitz, A. Kemper & B. Narasimhan, A survey of application generators, *IEEE Software*, januari 1985, pp. 40-54.

[HOL86] J.H. Holland *et al.*, *Induction: Processes of Inference, Learning, and Discovery*, MIT Press, 1986.

[JOH79] S.C. Johnson, YACC: Yet Another Compiler-Compiler, in: *UNIX Programmer's Manual*, Vol. 2B, 7th ed., Bell Laboratories, januari 1979, pp. 3-37.

[JSS85] N.D. Jones, P. Sestoft & H. Søndergaard, An experiment in partial evaluation: the generation of a compiler generator, in: *Rewriting Techniques and Applications, Lecture Notes in Computer Science*, Vol. 202, Springer-Verlag, 1985, pp. 124-140.

[KLI83] P. Klint, A survey of three language-independent programming environments, Rapport IW 240/83, Centrum voor Wiskunde en Informatica, Amsterdam, 1983.

[KOM82] H.J. Komorowski, Partial evaluation as a means for inferencing data structures in an applicative language: a theory and implementation in the case of PROLOG, in: *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, ACM, 1982, pp. 255-267.

[MEE86] L.G.L.T. Meertens, Algorithmics: Towards programming as a mathematical activity, in: J.W. de Bakker, M. Hazewinkel & J.K. Lenstra (eds), *Mathematics and Computer Science*, North-Holland, 1986.

[NIX85] R.P. Nix, Editing by example, *ACM Transactions on Programming Languages and Systems*, 7 (1985), 4, pp. 600-621.

[OOS82] C. Oostman, MISS: A machine independent simulation system, Rapport 051560-28<1982>08, Laboratorium voor Schakeltechniek en Techniek van de Informatieverwerkende Machines, TH Delft, 1982.

[PAU82] L. Paulson, A semantics-directed compiler generator, in: *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, ACM, 1982, pp. 224-233.

[SES85] P. Sestoft, The structure of a self-applicable partial evaluator, in: *Programs as Data Objects*, *Lecture Notes in Computer Science*, Vol. 217, Springer-Verlag, 1985, pp. 236-256.

[SNE85] G. Snelting, Experiences with the PSG — Programming System Generator, in: *Formal Methods and Software Development*, TAPSOFT Proceedings, Vol. 2, *Lecture Notes in Computer Science*, Vol. 186, Springer-Verlag, 1985, pp. 148-162.

[TOM86] M. Tomita, *Efficient parsing for natural language*, Kluwer Academic Publishers, 1986.

[WH81] P.H. Winston & B.K.P. Horn, *Lisp*, Addison-Wesley, 1981.