stichting
mathematisch
centrum

AFDELING INFORMATICA IN 21/82 DECEMBER
(DEPARTMENT OF COMPUTER SCIENCE)

H. KROEZE

EEN TAALONAFHANKELIJKE BENADERING VAN PRETTYPRINTEN

kruislaan 413 1098 SJ amsterdam

Printed at the Mathematical Centre, Kruislaan 413, Amsterdam, The Netherlands.

The Mathematical Centre, founded 11th February 1946, is a non-profit institution for the promotion of pure and applied mathematics and computer science. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

EEN TAALONAFHANKELIJKE BENADERING VAN PRETTYPRINTEN

a language-independent approach of prettyprinting

door

Henk Kroeze*

SAMENVATTING

Er wordt een complete methode gepresenteerd en geïmplementeerd, waarmee door een uitbreiding van de syntaxbeschrijving van een taal tekst in die taal van een mooie layout kan worden voorzien ("geprettyprint" kan worden). Voor de realisatie van dergelijke syntaxuitbreidingen is een beknopte verzameling gereedschap ontworpen. De methode is bovendien incrementeel van karakter, en derhalve in een interactieve omgeving (editing) te gebruiken.

TREFWOORDEN: Prettyprinting, programmeeromgevingen.

*Technische Hogeschool Twente, Enschede

INHOUD

1. Inleiding en onderzoeksopzet

2. De ontwikkelde verzameling prettyprint-primitieven

3. Fase 2: Van tussentekst naar layout

4. Fase 1: Van sourcetekst via (uitgebreide) syntaxdefinitie naar tussentekst

5. De taal HKL, een voorbeeld

6. Resultaten, ideeën voor uitbreiding en verder onderzoek

Referenties

Dit verslag beschrijft het resultaat van mijn stage aan de Afdeling Informatica van het Mathematisch Centrum gedurende de herfst van 1982. De stage werd begeleid door Jan Heering en Paul Klint. Ik wil hen bedanken voor hun steun en medewerking.

1. INLEIDING EN ONDERZOEKSOPZET

Om te beginnen wil ik wat begrippen introduceren en verduidelijken. Ten eerste, wat is prettyprinting? Zoals de naam al doet vermoeden is prettyprinting het op 2-dimensionale wijze "leesvriendelijk" weergeven van een in eerste instantie 1-dimensionale tekst, zodat de strukturen in die tekst duidelijk naar voren komen. Hierbij moet men vooral aan programmateksten denken. Overigens is met het begrip "leesvriendelijk" inderdaad enige niet te voorkomen subjectiviteit in de definitie aanwezig.

Het ligt voor de hand om een dergelijke prettyprinter te koppelen aan een tekstverwerker (editor) van een computersysteem. Zo'n editor kan er namelijk uitstekend gebruik van maken (mits de prettyprinter interactief is!), in de zin dat hij in staat is de hem aangeboden sourcetekst dan met een mooie layout op het scherm af te drukken, zonder dat de gebruiker zich daar verder om hoeft te bekommeren.

Als een editor de syntax van de hem aangeboden tekst kent, kan hij de gebruiker zelfs leiden bij het intypen ervan. Een dergelijke editor heet syntax-directed.

Nu houden Jan Heering en Paul Klint zich bezig met een project op het gebied van programmeeromgevingen. Een programmeeromgeving is een verzameling gereedschap voor het maken en bewerken van programmatuur, en binnen het project van Jan en Paul wordt een dergelijk systeem gebaseerd op taaldefinities [1].

Dit houdt in dat er in de programmeeromgeving meerdere gedefinieerde talen bestaan; een gebruiker communiceert op ieder moment in een van de talen uit het talenbestand met het systeem. Die communicatie verloopt via de boven beschreven syntax-directed editor/prettyprinter en deze gebruikers-interface is voor alle talen dezelfde, dat wil zeggen dat de interface universeel is omdat hij uitgaat van de taaldefinitie. Hieruit volgt dus in ieder geval dat het prettyprintgedeelte van de editor ook universeel dient te zijn! De aanwijzingen om voor iedere tekst in een taal tot een layout te komen, zullen een onderdeel van de taaldefinitie vormen. In figuur 1 is een en ander schematisch voorgesteld.

Samenvattend vinden we in de programmeeromgeving de volgende onderdelen die betrekking hebben op prettyprinting:

- Taaldefinities. Deze bestaan voor iedere taal uit:

  - Syntaxregels in bijv. de bekende BNF-notatie.

  - Semantiek-gedeelten bij de syntaxregels (uitgedrukt in een of andere in het talenbestand voorkomende taal).

  - "Prettyprintaanwijzingen", gelieerd aan of verweven met de syntaxregels.

- Een universele prettyprinter (als onderdeel van de editor).

In het hier beschreven onderzoek is, om dit beeld in te kunnen vullen, respectievelijk aandacht besteed aan:

1. De definitie van een algemene, maar beknopte en simpel te hanteren, verzameling taal-onafhankelijke prettyprintprimitieven. De primitieven uit die verzameling kunnen heel algemeen in een concrete tekst gebruikt worden om in die tekst bepaalde layoutvoorzieningen, voor het tekstgedeelte waar zij betrekking op hebben, aan te geven. Deze primitieven geven ons dan de mogelijkheid om de prettyprintaanwijzingen in de taaldefinities als volgt te realiseren: de aanwijzingen hoeven slechts aan te geven op welke syntactische structuren welke primitieven toegepast dienen te worden. De invoering van de primitieven is het onderwerp van hoofdstuk 2.

2. Beschrijving van een universele interactieve methode waarmee een concrete tekst in een willekeurige taal "geprettyprint" kan worden, onder uitvoering van genoemde aanwijzingen uit de taaldefinitie.

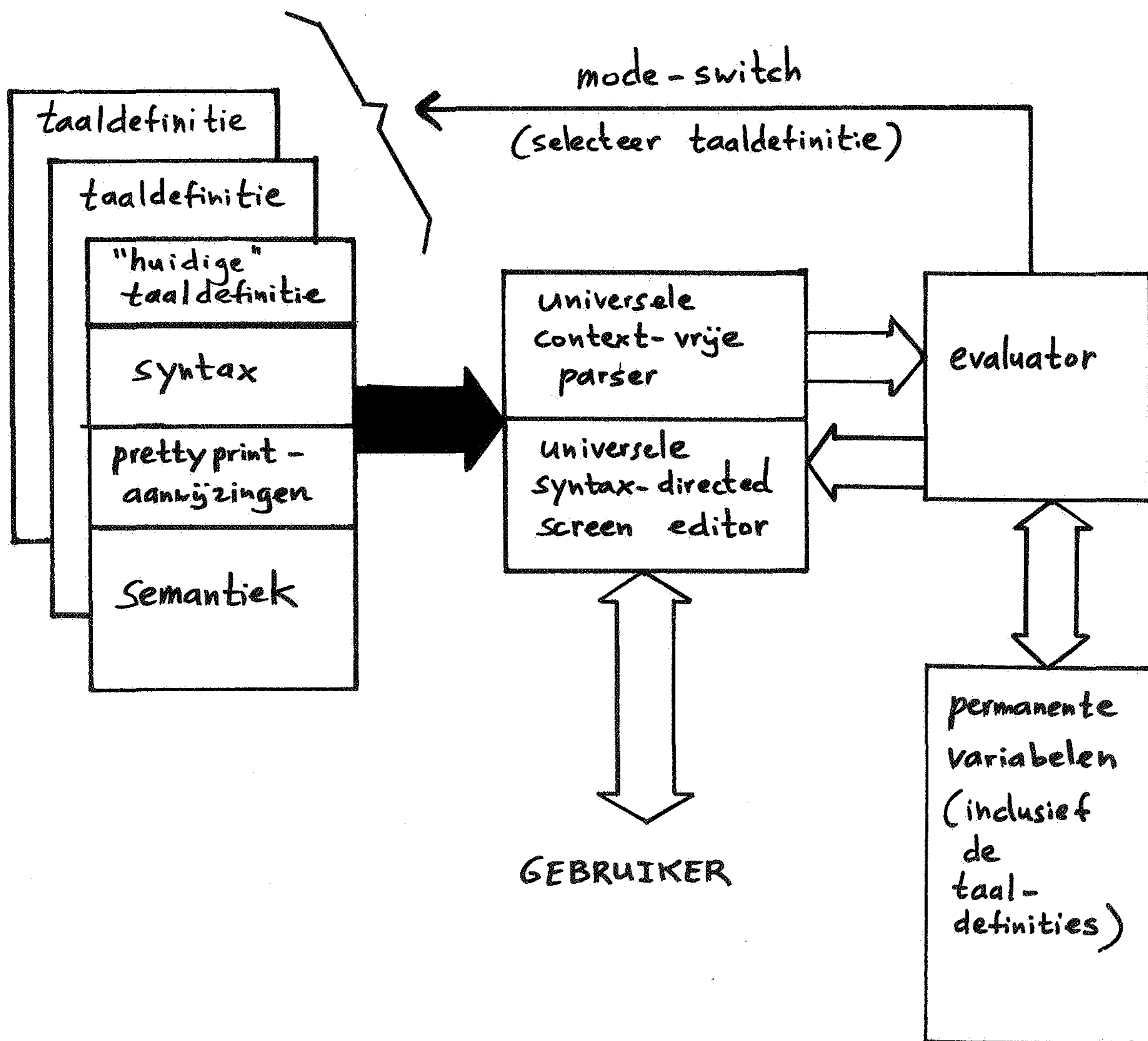

Fig. 1. Globale opzet van een programmeeromgeving gebaseerd op taaldefinities (vergelijk [1]).

Deze methode dient men zich conceptueel als volgt voor te stellen:

Er zijn 3 concurrente processen actief, te weten:

- Het intypen van tekst door de gebruiker.
- Het uitvoeren van de aanwijzingen uit de taaldefinitie, dat wil zeggen het incrementeel transformeren van de ingetypte stukken sourcetekst naar delen zogenaamde tussentekst door - zoals gezegd - prettyprint-primitieven toe te voegen. We zullen dit fase 1 noemen.
- Het afdrukken van tekstlayoutgedeelten uitgaande van stukken tussentekst. Dit laatste proces noemen we fase 2.

We merken op dat alleen fase 1 de taaldefinitie nodig heeft, terwijl fase 2 voor alle talen dezelfde is.

Als de communicatie van de 3 processen verloopt via buffers, dan ziet het geheel er globaal uit als geschetst in figuur 2.

De implementatie van een interactief bruikbaar algoritme voor fase 2 wordt vervolgens belicht in hoofdstuk 3, omdat in het onderzoek na het ontwikkelen van de prettyprint-primitieven als eerste aan fase 2 aandacht besteed is. Deze implementatie is gebaseerd op Oppen [2] en is geschreven in Summer, een op het Mathematisch Centrum ontwikkelde, zeer krachtige stringmanipulatietaal, die ook als "all round" programmeertaal toepasbaar is [5].

De manier waarop fase 1 algemeen voor een taal uitgewerkt wordt, alsmede de manier waarop prettyprintaanwijzingen bij de syntaxdefinitie in een taalbeschrijving ondergebracht kunnen worden, vormen het onderwerp van hoofdstuk 4.

Tot slot zal in hoofdstuk 5 een volledig uitgewerkt voorbeeld van het prettyprintprocédé behandeld worden voor het, om redenen van redelijke beknoptheid, speciaal hiervoor bedachte Pascal/Algol-achtige taaltje HKL.

2. DE ONTWIKKELDE VERZAMELING PRETTYPRINT-PRIMITIEVEN

Het effect van in een tekst geplaatste prettyprint-primitieven is, dat de tekst door die primitieven onderverdeeld wordt in kleinere stukken en dat elk van die stukken met een bepaalde layoutvoorziening wordt geassocieerd. Dit idee is ontleend aan Oppen [2].

Omdat we, vanwege de interactiviteit, de layout regel na regel incrementeel willen opbouwen, zijn er drie layoutvoorzieningen nuttig:

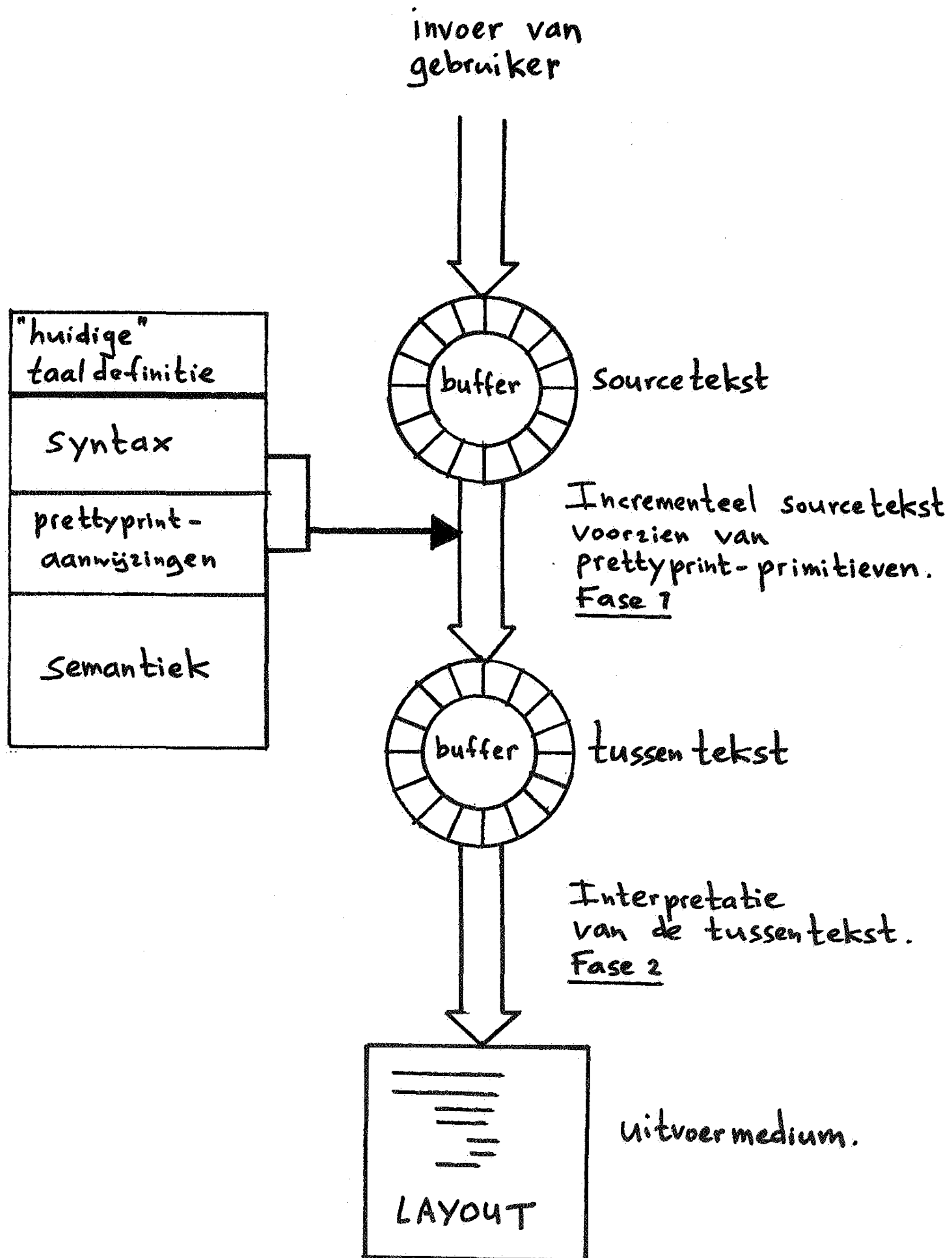

Fig. 2. Op interactieve basis van concrete tekst naar layout.

1. Als een stuk tekst op een nieuwe regel afgebroken dient te worden (omdat dat een betere layout geeft of omdat er geen tekst meer op past), willen we aan kunnen geven op welke positie van die nieuwe regel weer verder gegaan dient te worden (indentering). Dit om regels netjes onder elkaar te kunnen krijgen.

2. We willen kunnen aangeven dat een zeker stuk tekst niet afgebroken dient te worden wanneer dit te vermijden is. Dat wil zeggen dat het, indien het als geheel niet meer op de regel zou passen, in ieder geval op een nieuwe regel, waar meer ruimte is, gezet wordt. Dit om tekstgedeelten, die samen als een eenheid dienen te worden beschouwd, bij elkaar te kunnen houden.

3. We willen de mogelijkheid hebben om een verplichte spatie tussen gedeelten van een tekst aan te geven, alsmede de mogelijkheid om een verplichte overgang naar een nieuwe regel af te dwingen. Dit om scheiding van tekstgedeelten te kunnen realiseren.

Om die layoutvoorzieningen te realiseren, definieren we nu vier prettyprint-primitieven indentstream, blockstream, blank en indentreq. De tussentekst, dat wil zeggen de met prettyprint-primitieven "verrijkte" tekst, krijgt de volgende syntactische struktuur (BNF-notatie):

Tussentekst-syntax:

```
LEXICAL indentopen, indentclose, blockopen, blockclose,
        blank, indentreq, string.

<tussentekst>  ::= <indentstream>.
<indentstream> ::= <indentopen> <string> ( <separator> <stream> )*
                   <indentclose>.
<separator>    ::= <blank> | <indentreq>.
<stream>       ::= <indentstream> | <blockstream> | <string>.
<blockstream>  ::= <blockopen> <string> ( <separator> <stream> )*
                   <blockclose>.
```

Opmerkingen:

LEXICAL definieert een reeks terminalsymbolen die als lexicale eenheden beschouwd worden; in dit geval is een string een (niet lege) verzameling afdrukbare symbolen, en zijn de andere lexicale eenheden symbolen voor de prettyprint-primitieven.

* is een repetitie-indicator, en geeft aan dat de voorafgaande term 0 of meer keren herhaald kan voorkomen.

De betekenis van de primitieven indentstream, blockstream, blank en indentreq definiëren we als volgt:

indentstream: Als deze <stream> op een nieuwe regel afgebroken moet worden, dan wordt er geïndenteerd tot de positie zoals die gold bij het passeren van het <indentopen>-symbool Alles komt dus op eenzelfde lijn onder de eerste <string> uit deze <stream> te staan indien er afgebroken moet worden. (1e voorziening).

blockstream: Als deze <stream> helemaal op de huidige regel past, is er niets aan de hand. Zo niet dan wordt er eerst geïndenteerd tot de positie zoals die bepaald is door de <indentstream> waarvan deze <blockstream> deel uitmaakt. (2e voorziening).

blank: Als een <blank> gevolgd wordt door een <blockstream>, bepaalt deze laatste wat er gebeurd: de <blank> is een spatie wanneer spatie gevolgd door <blockstream> nog op de regel passen, de <blank> heeft geen betekenis wanneer vanwege de <blockstream> indentatie plaatsvindt.
In beide andere gevallen (opvolger is <string> of <indentstream>) bepaalt de eerstvolgende <string> na de <blank> wat er gebeurt: een spatie als daarna deze <string> ook nog op de regel past, niets wanneer een indentatie nodig is. (3e voorziening).

indentreq: Er wordt altijd geïndenteerd (speciale blank!). (3e voorziening).

We merken op dat dus alle acties met betrekking tot het afbreken van tekst op nieuwe regels door een <separator> verzorgd worden.

De onderstaande notatie is bedacht om prettyprint-primitieven redelijk weer te kunnen geven in een concrete tussentekst, zonder dat uitschrijven daarvan tot onoverzichtelijkheid leidt.
Zo wordt de representatie van

indentopen.......indentclose
een onderlijning met ____________________,

blockopen........blockclose
een bovenblok ┬──────────────────┬,

het symbool voor een blank het tekentje ␣,

en het symbool voor een indentreq tenslotte een kruisje x.

Als afsluiting van dit hoofdstuk wil ik een voorbeeldje geven van het gebruik van de boven gedefinieerde primitieven.

Beschouwen we een willekeurig stukje tekst, waarin 2 alternatieven

en specificatie van die alternatieven voorkomen, zoals

NU: OF -LETTERS: A E, -KLANKEN: AI EI

We maken van deze tekst op de volgende wijze een tussentekst (waarbij het geheel uiteraard voldoet aan de tussentekst-syntax):

```
                                          ┌──────────────────────────────┐
NU: x OF u - u LETTERS: u ┌A u E, ┌ u │ - u KLANKEN: u ┌AI u EI ┐ │
               ---------------------      ----------------------
           ---------------------------------------------------------
-----------------------------------------------------------------------
```

Wat we hiermee dus volgens de eerder gegeven semantiek aangeven, is:

- De stukken tekst "A E,", "-KLANKEN: AI EI" en daarbinnen "AI EI", willen we als geheel bij of onder elkaar houden.
- Als de tekst "LETTERS: A E," afgebroken moet worden, dan wordt er geïndenteerd tot onder de positie van "LETTERS".
- Evenzo, als de tekst "KLANKEN: AI EI" afgebroken moet worden, dan wordt er geïndenteerd tot "KLANKEN".
- Moet "-LETTERS: A E, -KLANKEN: AI EI" afgebroken worden zonder dat deze afbreking valt binnen bovengenoemde twee deelstukken, dan wordt er geïndenteerd tot de eerste "-".
- Evenzo voor de gehele tekst; bij een afbreking anders dan in een van de genoemde deelstukken wordt er verder gegaan direct onder "NU:".
- Na "NU:" willen we dat de tekst altijd afgebroken wordt.
- De plaatsing van de u-tjes verteld dat alleen daar een spatie, of anders een afbreking in de tekst, kan voorkomen.

Dit levert bij verschillende regelbreedtes de volgende layout:

(regelbreedte 17)

```
NU:
OF - LETTERS:
     A E,
   - KLANKEN:
     AI EI
```

(regelbreedte 18)

```
NU:
OF - LETTERS: A E,
   - KLANKEN:
     AI EI
```

(regelbreedtes 19 tot en met 34)

```
NU:
OF - LETTERS: A E,
   - KLANKEN: AI EI
```

(regelbreedtes 35 of groter)

```
NU:
OF - LETTERS: A E, - KLANKEN: AI EI
```

3. FASE 2: VAN TUSSENTEKST NAAR LAYOUT

In dit hoofdstuk wil ik de implementatie van de semantiek bij de in het vorige hoofdstuk gedefinieerde tussentekst kort beschrijven. Tevens zullen twee nuttige uitbreidingen bij die semantiek worden ingevoerd.

Noemen we de verschillende lexicale symbolen (zie vorige hoofdstuk) in een tussentekst (dus blockopens, blockcloses, indentopens, indentcloses, blanks, indentreqs en strings) tokens, dan berust het principe voor het interpreteren van de primitieven op het aan de tokens toekennen van zogenaamde geassocieerde lengtes [2].

We definiëren:

- De geassocieerde lengte van een string is gelijk aan het aantal symbolen waaruit die string bestaat.

- De geassocieerde lengte van een blockclose of indentclose, mits deze direct na een string volgt, is gelijk aan 1 + het aantal block- en/of indentcloses dat direct op de close volgt.

- De geassocieerde lengte van een blank is afhankelijk van door wat voor token de blank gevolgd wordt (zie hoofdstuk 3). Is dat een blockopen, ten teken dat er een blockstream volgt, dan is de lengte gelijk aan 1 + de som van de geassocieerde lengten van alle strings + het aantal blanks in die blockstream. Is dat echter geen blockopen dan geldt voor de geassocieerde lengte dat deze gelijk is aan 1 + de geassocieerde lengte van de eerstvolgende string.

- De geassocieerde lengte van een indentreq is gelijk aan "oneindig".

Alle andere tokens hebben geen geassocieerde lengte (nul).

Als we nu in figuur 2 fase 2 opsplitsen in 2 concurrente processen, die allebei op de tussentekstbuffer (tustbuf) opereren, dan kan een van die processen tokens in tustbuf voorzien van hun geassocieerde lengtes (associate), terwijl de andere al van een lengte voorziene tokens uit tustbuf kan halen en kan afdrukken. Immers, als tokens voorzien zijn van hun geassocieerde lengte is op onderstaande wijze eenvoudig de layout te realiseren (advanceprint):

Haal een token uit tustbuf.

Als het token een indentopen of blockopen is zet dan de huidige nog beschikbare ruimte op de regel (space) op een stack (indentstack).

Als het een indentclose of een blockclose is, verwijder dan zoveel stackwaarden van de top van de stack als aangegeven wordt door de geassocieerde lengte.

Als het een blank of indentreq is dan: is de geassocieerde lengte kleiner of gelijk aan space, druk dan een spatie af, haal anders de nieuwe waarde voor space van de top van de stack (zonder die te verwijderen!), en indenteer op een nieuwe regel tot die spacewaarde.

Als het een string is, druk deze dan af.

Doe dit achtereenvolgens voor alle tokens.

Associate berekent zoals gezegd de geassocieerde tokenlengtes. Bij het berekenen van lengtes voor blanks voor een blockstream maakt ook deze procedure gebruik van een stack (blockstack). Hier wordt namelijk de index in tustbuf van het blockopentoken bewaard, om zodra de bijbehorende blockclose wordt gelezen de lengte voor de blank, in de buffer in te vullen en deze blank vrij te geven voor verwerking door advanceprint.

Als implementatievorm voor tustbuf is de circulaire buffer met twee bufferpointers, om aan te kunnen geven welk gedeelte gevuld is, gebruikt. De stacks zijn geïmplementeerd als speciale circulaire stacks, met 2 stackpointers om zowel van de top als van de onderkant gemakkelijk elementen te kunnen verwijderen. Het speciale eraan is het feit dat voor elke stackplaats een teller aanwezig is om aan te geven hoe vaak een element op die plaats staat. Wordt namelijk een element op de stack aangeboden dat al op de top staat, dan hoeft slechts de betreffende teller met 1 opgehoogd te worden, en omgekeerd bij het verwijderen van elementen. Dit spaart stackruimte indien bijv. vaak dezelfde indentatie op de indentstack gezet wordt.

Dé interactiviteit wordt in het beschreven procédé gewaarborgd door

invoering van twee tellers rightotal en leftotal, die respectievelijk aangeven het aantal printbare characters (dus afkomstig van tokens blank en/of string) dat door associate in tustbuf al behandeld is, en het aantal characters dat er door advanceprint al weer uitgehaald en verwerkt is. Het verschil rightotal-leftotal geeft dus op elk moment aan hoeveel characters nog niet zijn vrijgegeven, omdat bepaalde lengtes nog niet berekend zijn. Als nu op een gegeven moment dit verschil groter wordt dan de nog beschikbare ruimte (space), dan kan de blank voor de blockstream die de verwerking "ophoudt" omdat zijn lengte berekend wordt, rustig lengte oneindig toegewezen krijgen. Dit, omdat berekening van de werkelijke lengte slechts tot hetzelfde resultaat, een indentering, zou leiden [2].

We kunnen dus stellen dat de mate van interactiviteit (respons) evenredig is met de uitvoerregelbreedte die ter beschikking is (margin). Gemiddeld zullen er namelijk margin/2 karakters in tokens aangeboden moeten worden voordat er (weer) iets geprint kan worden.

De voorgaande beschouwing geeft ons bovendien de mogelijkheid een grootte voor tustbuf vast te stellen, die in ieder geval voldoende is. Immers, de enige reden waarom de buffer met tokens vol gaat lopen is, omdat de lengte van een blank, die gevolgd wordt door een blockstream, bepaald moet worden. In dat geval komen er namelijk steeds tokens bij, terwijl advanceprint geen tokens kan verwijderen. Maar het verschil rightotal-leftotal neemt steeds toe als de bij de blockstream behorende blockclose voorlopig uitblijft, omdat er dan tokens volgen waaronder zich in ieder geval strings, blanks en/of indentreq's bevinden (zie tussentekstsyntax). Als dit verschil te groot wordt, zal de blank zoals boven beschreven voortijdig worden "doorgestart", waarna dus tevens door advanceprint weer bufferruimte vrijgemaakt zal worden. In het meest ongunstige geval zijn derhalve hooguit margin bufferplaatsen nodig per soort token dat kan volgen. Daarvan zijn er 4 (blockopen/indentopen, string, blank/indentreq, blockclose/indentclose), zodat er in totaal margin*4 ruimte nodig is.

Als grootte voor de blockstack (zie boven) is de waarde van margin voldoende, want er kunnen in geen geval meer dan margin blockopens tegelijkertijd genest actief zijn. Zouden er meer volgen door een diepere nesting, dan zal de blank bij de oudste blockopen doorgestart worden en de oudste blockstackplaats weer worden vrijgemaakt, omdat iedere blockopen minstens gevolgd wordt door een string en derhalve weer rightotal-leftotal groter wordt dan space.

De maximale grootte van de indentstack is niet statisch te bepalen, omdat vanwege het toe te passen wrapping mechanisme (zie onder bij tweede semantiek-uitbreiding) en vanwege het feit dat een tussentekst uit een ongelimiteerde streamnesting kan bestaan het aantal te onthouden indentatiewaarden ook ongelimiteerd is! Door de indentstack dezelfde grootte te geven als tustbuf, is er meestal voldoende speling met betrekking tot nesting.

Om de invoering van een eerste uitbreiding van de tussentekstsemantiek te rechtvaardigen beschouwen we het volgende voorbeeld:

We nemen als tussentekst

```
IF u a>b u | THEN u BEGIN u X1:=0; u X2:=0 u END |
                            -------------
                     ---------------------------
------------------------------------------------
```

Als de ruimte om deze stream af te drukken groot genoeg is, is er niks aan de hand, alles past op dezelfde regel. Is er echter maar bijvoorbeeld een ruimte van 20 characters dan zou het resultaat er als volgt uitzien:

```
IF a>b
THEN BEGIN X1:=0;
           X2:=0 END
```

Maar dit is duidelijk niet wat we willen: omdat er tussen de strings 'X1:=0;' en 'X2:=0' in de binnenste indentstream van de tussentekst een indentatie naar een nieuwe regel (break) heeft plaatsgevonden, willen we dat de string END onder de string BEGIN geïndenteerd wordt. Oftewel, we zouden willen dat zo'n break binnen een stream direct na afloop van die stream ervoor zorgt dat er ook dan een break plaatsvindt. Deze uitbreiding blijkt in het algemeen onmisbaar te zijn en is als volgt te definiëren:

De eerste string (als er een is) na de close van een stream S, waarin een break is opgetreden, wordt altijd geïndenteerd op dezelfde positie als die van de eerste string uit de omvattende stream, dat wil zeggen de stream waarin S zich bevindt.

N.B. Overigens is met deze uitbreiding het effect altijd nog te vermijden. In het voorbeeld door de tekst als volgt van primitieven te voorzien:

```
IF u a>b u | THEN u BEGIN u X1:=0; u X2:=0 u END |
                            ---------------------
------------------------------------------------
```

De noodzaak van een tweede uitbreiding heeft betrekking op het volgende: na al een indentering kan het zijn dat de eerstvolgende string uit een stream nog steeds niet op de regel past, omdat de string gwoon te groot is. Het maken van de layout loopt rechts "van het papier" af.

Besloten is om dan weer aan het begin op positie 1 van de regel verder te gaan met afdrukken (wrap around); maar zodra de stream waarvoor dit noodzakelijk is eindigt, kan o.a. met behulp van de boven beschreven eerste uitbreiding weer teruggegaan worden naar het oude indentatieniveau. Wrap around zorgt ervoor dat prettyprinting altijd mogelijk is, terwijl toch voldoende wordt aangegeven dat er iets bijzonders aan de hand is, en herschrijven van de tekst door de gebruiker (een niet goed gestructureerd programma!) misschien beter is! [4]

Tot slot, een volledig uitgewerkte en gedocumenteerde listing van het Summer-programma, dat fase 2 kan uitvoeren voor een tussentekst in een file, is beschikbaar. Dit programma vraagt margin, de te definiëren uitvoerregelbreedte, als parameter, zodat layouts voor verschillende breedtes gemaakt kunnen worden.

4. FASE 1: VAN SOURCETEKST VIA (UITGEBREIDE) SYNTAXDEFINITIE NAAR TUSSENTEKST

De prettyprintaanwijzingen in een taaldefinitie dienen voor sourcetekstgedeelten aan te geven welke van de prettyprintprimitieven daarop toegepast moeten worden. Nu wordt de opbouw van sourceteksten in de syntax beschreven, dus ligt het voor de hand de prettyprint-primitieven te "verweven" met de syntaxregels. Een dergelijke syntax noemen we <u>uitgebreid</u>.

In een typische programmeertaal kan bijvoorbeeld een procedure-aanroep van de volgende vorm voorkomen (BNF-notatie):

```
<call> ::= <name> ´(´ <exp> ( ´,´ <exp> )* ´)´.
<exp>  ::= <name> ´+´ <name>.
```

De layout voor procedure-aanroepen "weven" we nu bijvoorbeeld als volgt in de syntaxregels (we gebruiken de eerder beschreven notatie):

```
                 ______________________________________________
                |                                              |
<call> ::= <name> u | ´(´ u <exp> ( u ´,´ u |‾<exp>‾| )* u ´)´ |.
                        ---------------------------------
           ------------------------------------------------------


<exp>  ::= <name> u ´+´ u <name>.
```

Dit betekent het volgende: Er verschijnen alleen spaties of eventueel afbrekingen tussen <name> en ´(´, tussen ´(´ en de eerste <exp>, tussen iedere <exp> en een eventueel volgende ´,´ alsmede tussen de ´,´ en de daarop volgende <exp>, en tussen de laatste <exp> en ´)´.
Alles wat tussen haakjes staat, inclusief de haakjes zelf, dient bij el-

kaar te blijven, d.w.z. moet in eerste instantie helemaal achter <name> passen en als dat niet mogelijk is dan moet het afdrukken ervan direct onder <name> beginnen.
Ook een <exp> moet altijd in eerste instantie in zijn geheel op een regel passen. Als dat niet kan dan moet weer direct onder de eerste <exp> begonnen worden; past de sluithaak niet meer op de regel, dan wordt ook deze uitgelijnd met de eerste <exp>.
Binnen een <exp> verschijnen spaties (en is eventueel afbreking mogelijk) voor en na het ´+´-teken.

De tussentekst volgt direct uit de uitgebreide syntaxregel door in een concrete tekst die het produkt is van zo´n regel, de prettyprintprimitieven te plaatsen zoals de uitgebreide regel dit voorschrijft. Alles wat door de uitgebreide regel kan worden geproduceerd dient echter aan de tussentekst-syntax te voldoen. In het voorbeeld is dit slechts het geval indien <name> met een string begint of een string is!

In ons voorbeeld zou de tussentekst voor de zin

PRINT(A+B,C+D,E+F)

er uitzien als:

```
         ┌────────────────────────────────────────────────┐
PRINT u │ ( u A u + u B u , u ┌C u + u D┐ u , u ┌E u + u F┐ u ) │
              ------------------------------------------------
-----------------------------------------------------------------
```

Deze tussentekst kan zoals we gezien hebben door fase 2 verwerkt worden. Resteert ons echter een algoritme te bedenken, dat uitgaande van de uitgebreide syntax en een concrete tekst in de bijbehorende taal de tussentekst oplevert.
Voordat hierop ingegaan wordt, willen we eerst een laatste algemene uitbreiding van de tussentekstsyntax met bijbehorende semantiek introduceren.

Wanneer we de verkregen tussentekst uit het voorbeeld zouden interpreteren bij een regelbreedte 8 krijgen we

```
PRINT
( A + B
 ,
 C + D
 ,
 E + F
 )
```

Dit komt omdat in de uitgebreide regel voor <call> voor de ',' in de argumentenlijst een blank staat, zodat afbreking op die plaats geoorloofd is! Die blank mag daar echter niet weggelaten worden, hij moet daar staan om aan de tussentekstsyntax te voldoen, aangezien namelijk de string ',' door herhaling(en) direct achter een blockstream kan komen te staan, en dus volgens de definitie van tussentekst door een separator daarvan gescheiden moet zijn.

De oplossing voor dit probleem is uitbreiding van het begrip separator. We herdefiniëren dit begrip als volgt:

<separator> ::= [<string>] (<blank> | <indentreq>).

De bijbehorende semantiek is: indien een separator begint met een string, dan wordt deze string beschouwd als (nog) horend bij de laatst ervoor gelezen string, en zal dáaraan vastgekoppeld en van de separator afgehaald worden. Uiteraard wordt bij de berekening van de geassocieerde lengtes daarmee rekening gehouden.
Deze betekenis transformeert de nieuwe separatorregel dus eigenlijk weer tot

<separator> ::= <blank> | <indentreq>.

zodat voorgaande implementatiebeschouwingen uit hoofdstuk 3 over buffergroottes etc. nog steeds geldig zijn!

Zo kan dus, om op bovenstaand voorbeeld terug te komen, de storende blank voor de ',' nu weggelaten worden. Hierdoor wordt de combinatie van ',' en de blank voor de blockstream tot nieuwe separator, terwijl bij interpretatie de ',' vastgemaakt wordt aan de laatste string uit de voorafgaande <exp>.
Een afbreking tussen <exp> en ',' is derhalve onmogelijk geworden, zodat het resultaat voor breedte 8 nu wordt

```
PRINT
( A + B,
  C + D,
  E + F
  )
```

We moeten zoals gezegd nog een algoritme bedenken dat uitgaande van de uitgebreide syntax en een concrete tekst in de bijbehorende taal de tussentekst oplevert.
Het concrete resultaat van zo'n algemeen algoritme zal in staat moeten zijn sourcetekst te ontleden, om dan bij herkenning van syntactische categoriën in die tekst op de juiste plaatsen prettyprintprimitieven tussen te voegen, zoals dat in de uitgebreide syntax vastgelegd is. Vanwege de beperkte tijd is geen aandacht besteed aan een universele uitwerking hiervan, maar is gebruik gemaakt van een bestaande, op het MC ontwikkelde

en geïmplementeerde parsergenerator PGEN [6]. PGEN verwacht als invoer de taalsyntax, opgeschreven in de in dit verslag al meerdere malen gebruikte uitgebreide BNF-notatie.

PGEN genereert op grond van deze syntax de parser voor de taal als een Summerprogramma. De verkregen parser zal in ieder geval automatisch de controle op het syntactisch correct zijn van de invoertekst verrichten, en geschikte foutmeldingen, indien nodig, genereren.

Hoe echter, kunnen we er voor zorgen dat de parser ook iets meer zal doen dan slechts het herkennen van de invoertekst? Daartoe biedt PGEN ons de volgende mogelijkheid. In de mee te geven syntaxregels mogen ook zogenaamde labels aanwezig zijn. Die labels worden ieder geïdentificeerd met een stukje programmatuur (in Summer) dat onder de betreffende regel toegevoegd wordt. Bijvoorbeeld:

```
<if-stmt> ::= IF <expr> /L1/ THEN /L2/ <block> /L3/.

/L1/: xp:=evalueer(expr);
/L2/: blokteller:=blokteller+1;
/L3/: if xp then evalueer(block);
      blokteller:=blokteller-1;
```

Doordat PGEN ook deze stukjes semantiek aan de parser toevoegd, wordt er voor gezorgd dat zij tijdens het parsen geëxecuteerd worden op de door de labelplaatsing gedefinieerde momenten.

We zien dus dat we een implementatie voor fase 1, met PGEN als volgt kunnen verkrijgen:

Gegeven de uitgebreide syntax; transformeer die (weer) naar de oorspronkelijke syntax zonder prettyprint-aanwijzingen, maar voeg nu overal waar een primitief staat (blank, indentreq) of begint/eindigt (stream) een label toe; identificeer zo'n label met een simpel stukje Summer dat een symbool voor de betreffende prettyprintinstructie wegschrijft.
N.B. het is precies dit "handwerk" dat aangeeft dat de PGEN-oplossing niet helemaal algemeen fase 1 oplost (zie ook hoofdstuk 6).

Voeg juist na alle terminalsymbolen (lexicals, keywords van de syntax, characterstrings in de syntax) ook een label toe, en associeer met deze labels een eveneens simpel Summerstatement dat het zojuist gelezen symbool wegschrijft.

De gegenereerde parser zal dan bij gebruik dus alle gelezen tekst weer wegschrijven, maar ook op de door de uitgebreide syntax bepaalde plaatsen, de aangegeven primitieven toevoegen. Kortom, hij genereert de gevraagde tussentekst.

Ik wil dit hoofdstuk besluiten met het uitvoeren van dit PGEN-schema voor ons voorbeeldje.

```
LEXICAL name.
<call> ::= <name> '(' <exp> ( ',' <exp> )* ')'.
<exp> ::= <name> '+' <name>.
```

Als uitgebreide taalsyntax was ingevoerd

```
LEXICAL name.
                   ________________________________________________
                  |                                                |
<call> ::= <name> u | '(' u <exp> ( ',' u | <exp> | )* u ')' |.
                                --------------------------------
                -----------------------------------------------------

<exp> ::= <name> u '+' u <name>.
```

De PGEN syntaxinvoer wordt derhalve

```
LEXICAL name.

<call> ::= /L1/ <name> /L2/ '(' /L3/ <exp>
           (',' /L4/ <exp> /L5/ )* /L6/ ')' /L7/.

/L1/: out.put(indentopen);
/L2/: out.put(sy,blank,blockopen);
/L3/: out.put(sy,blank,indentopen);
/L4/: out.put(sy,blank,blockopen);
/L5/: out.put(blockclose);
/L6/: out.put(blank);
/L7/: out.put(sy,indentclose,blockclose,indentclose);


<exp> ::= <name> /L8/ '+' /L9/ <name> /L10/.

/L8/: out.put(sy,blank);
/L9/: out.put(sy,blank);
/L10/: out.put(sy);
```

Hierin is 'sy' een globale variabele die PGEN ter beschikking stelt, en die steeds na het lezen van een terminalsymbool gedurende het parsingsproces dat symbool bevat. Het Summerstatement out.put() bete-

kent: schrijf de argumenten weg naar de file 'out'. De symbolen 'indentopen', 'blockopen', etc. zijn globale constanten, en deze kunnen ook aan PGEN worden meegegeven, waarna de declaratie ervan in de gegenereerde parser verwerkt wordt.

Krijgt nu de gegenereerde parser een instantie van <call> als invoer, dan maakt deze de gehele tussentekst aan in de file 'out'.
We zien dus dat het geschetste schema helaas nog niet direct een incrementele parser oplevert.

Tenslotte wil ik opmerken dat uit de in dit hoofdstuk beschreven uitwerking voor fase 1 volgt dat prettyprinten alleen mogelijk is voor stukken tekst die syntactisch gedefinieerd zijn. Bij een heleboel programmeertalen is dat niet het geval voor commentaar. Zo mag bijv. volgens het Pascalrapport overal in een Pascaltekst commentaar voorkomen, en bij verwerking van zo'n tekst wordt commentaar op lexicaal niveau al weggefilterd.
Wil men echter commentaar ook meenemen bij prettyprinten, en dat is mijns inziens absoluut een vereiste, dan moet het als syntactische categorie in de grammatica ingevoerd worden.

5. DE TAAL HKL, EEN VOORBEELD.

Voor een demonstratie van het ontworpen prettyprintschema is HKL bedacht, een taal met Pascal/Algol-achtige constructies. Hieronder volgt allereerst de syntax voor HKL, beschreven in uitgebreid BNF.

```
LEXICAL ident,   #identifier#
        integer, #unsigned number#
        word,    #serie ASCII-karakters (behalve spatie of '#')
                  in een commentaar#
        name.    #identifier gevolgd door '(' #

<hkl-program> ::= <header> [ <comment> ] [ <declarations> ]
                  <block>.

<header> ::= HKLPROGRAM <ident> ';'.

<comment> ::= '#' <word> ( <word> )* '#'.

<declarations> ::= VAR <ident> ( ',' [ <comment> ] <ident> )* ';'
                   [ <comment> ].

<block> ::= ( BEGIN <statement-list> END ) | <simple-statement>.

<statement-list> ::= [ <comment> ] <statement>
                     ( ';' [ <comment> ] <statement> )*.

<statement> ::= <complicated-statement> | <simple-statement>.
```

```
<complicated-statement> ::= <if-statement> | <while-statement>.

<if-statement> ::= IF <expression> THEN <block> [ ELSE <block> ].

<while-statement> ::= WHILE <expression> DO <block>.

<simple-statement> ::= ( <procedure-call> [ <comment> ] )
                     | <assignment>.

<procedure-call> ::= <name> ´(´ [ <arg-list> ] ´)´.

<arg-list> ::= <expression> ( ´,´ <expression> )*.

<assignment> ::= <ident> ´:=´ <expression>.

<expression> ::= <term> [ <comment> ]
                 ( <adding-op> <term> [ <comment> ] )*.

<term> ::= [ <comment> ] <factor>
           ( <multi-op> [ <comment> ] <factor> )*.

<factor> ::= ( ´(´ <expression> ´)´ )
           | <variable>
           | <procedure-call>.

<variable> ::= <ident> | <integer>.

<adding-op> ::= ´+´ | ´-´.

<multi-op> ::= ´*´ | ´/´.
```

We lezen uit de syntax dat commentaar een syntactische categorie is. Deze is op te vatten als een verzameling ASCII-tekens (uitgezonderd het ´#´-teken), die op te delen is in woorden gescheiden door ASCII-spaties. HKL staat een tamelijk vrij gebruik van commentaar toe, ondanks het feit dat commentaar op syntactisch en niet op lexicaal niveau behandeld wordt.

Men moet wel bedenken dat bij de introduktie van HKL geen aandacht aan de semantiek geschonken is (een HKL-programma doet wat een Pascal/Algol-kenner denkt dat het doet), omdat die voor het prettyprinten van HKL-teksten verder onbelangrijk is.

De volgende stap is, zoals we gezien hebben, de HKL-syntax te voorzien van prettyprintaanwijzingen (hoofdstuk 4). Het ontwerpen van deze uitgebreide syntax heeft onderstaand resultaat opgeleverd:

```
1. LEXICAL ident,integer,word,name.

2. <hkl-program> ::= <header> x [ <comment> x ]
                     --------------------------

                     [ <declarations> x ] <block>.
                     ----------------------------

3. <header> ::= HKLPROGRAM u <ident> ´;´.

4. <comment> ::= ´#´ u <word> ( u <word> )* u ´#´.
                       --------------------

5. <declarations> ::= VAR u <ident> ( ´,´ u [ | <comment> | x ]
                            -----------------------------------

                      <ident> )* ´;´ [ u | <comment> | ].
                      ----------------------------------

6. <block> ::= ( BEGIN u <statement-list> u END ) | <simple-statement>.
                ------------------------------      ------------------

7. <statement-list> ::= | [ <comment> x ] <statement>

                      ( ´;´ u [ | <comment> | x ] | <statement> | )* |.

8. <statement> ::= <complicated-statement> | <simple-statement>.

9. <complicated-statement> ::= <if-statement> | <while-statement>.

10. <if-statement> ::= IF u <expression>
                            ------------

                       u | THEN u <block> [ u | ELSE u <block> | ] |.
```

```
11. <while-statement> ::= WHILE u <expression> u | DO u <block> |.

12. <simple-statement> ::= ( <procedure-call> [ u | <comment> | ] )
                           | <assignment>.

13. <procedure-call> ::= <name> u | ´(´ u  [ <arg-list> u ] ´)´ |.

14. <arg-list> ::= <expression> ( ´,´ u | <expression> | )*.

15. <assignment> ::= <ident> u ´:=´ u <expression>.

16. <expression> ::= <term> [ u | <comment> | ]

                   ( u <adding-op> u | <term> | [ u | <comment> | ] )*.

17. <term> ::= [ <comment> x ] <factor>

               ( u <multi-op> u [ | <comment> | x ] | <factor> | )*.

18. <factor> ::= ( ´(´ u <expression> u ´)´ )

                 | <variable>
                 | <procedure-call>.

19. <variable> ::= <ident> | <integer>.

20. <adding-op> ::= ´+´ | ´-´.

21. <multi-op> ::= ´*´ | ´/´.
```

De overwegingen bij bovenstaand ontwerp zal ik, voor bepaalde regels, nog even behandelen:

ad 2. De gehele tussentekst moet een indentstream zijn (zie N.B. hieronder). Na de <header> en ook na een eventueel <comment> en <declarations> wil ik dat het bijbehorende <block> op regelpositie 1 begint.

ad 4. Als de <word>s van een <comment> niet meer op een regel passen indenteren we bij het verdergaan op een nieuwe regel onder het eerste <word>. Zo'n break in een <comment> werkt door vlak voor het commentsluitteken '#' (zie hoofdstuk 3), zodat wanneer het commentopenteken '#' op de indenteringspositie van de stream, waarbinnen zich de <comment> bevindt, staat, het sluitteken hieronder gezet wordt.

ad 5. De <declarations>-lijst wordt in geval van een break uitgelijnd op de positie van de eerste <ident>. Van een eventueel <comment> in de lijst eisen we dat hij in zijn geheel achter een <ident> moet passen, kan dat niet dan begint het <comment> ook geïndenteerd op die positie. Na een <comment> wordt altijd weer op die positie begonnen.

ad 6.,7. De <statement-list> moet in zijn geheel achter de BEGIN van een <block> passen, kan dat niet dan wordt er onder de BEGIN mee begonnen. <Statement>s in de <statement-list> moeten ook in eerste instantie geheel op een regel passen, zo niet dan beginnen ook zij op het indentatieniveau van BEGIN. Voor een <comment> geldt in dit geval mutatis mutandis hetzelfde als voor een <comment> in een <declarations-list>.

ad 10.,11. De <expression> wordt eventueel uitgelijnd op het begin ervan, de THEN- en ELSE-gedeelten moeten één geheel blijven, evenals het DO-gedeelte van het WHILE-statement.

ad 13.,14. Zie het voorbeeld in hoofdstuk 4.

ad 16.,17. Ook hier is sprake van uitwerking van eisen met betrekking tot het één geheel blijven van respectievelijk <term>s, <factor>s en wederom <comment>s.

ad 18. Voor de <expression> en de sluithaak geldt weer dat we willen dat eventueel uitlijning op het begin van de <expression> geschiedt.

N.B. Regel 2 (de startregel) is een indentstream. Er is ook voor gezorgd dat iedere stream met een string (een van de lexicals of een van de taalsymbolen) begint dan wel een string is, en bovendien met een stream eindigt. Tussen twee streams staat steeds een separator. Om deze redenen voldoet de uitgebreide syntax, en voldoen alle zinnen die deze kan produ-

ceren, aan de tussentekstsyntax!

PGEN kan nu de implementatie van fase 1 genereren zoals beschreven in hoofdstuk 4. De uitgebreide syntax wordt daartoe eerst volgens de daar gegeven directe, stelselmatige omzetting getransformeerd naar PGEN-invoer. De omzetting ziet er bijvoorbeeld voor regel 6 en 7 als volgt uit:

```
<block> ::= ( /s220/ BEGIN /s230/ <statement-list> /s240/ END /s250/ )
          | ( /s260/ <simple-statement> /s270/ ).

/s220/: out.put(indentopen);
/s230/: out.put(sy,blank);
/s240/: out.put(blank);
/s250/: out.put(sy,indentclose);
/s260/: out.put(indentopen);
/s270/: out.put(indentclose);

<statement-list> ::= /s280/ [ <comment> /s290/ ] <statement>
                     ( ';' /s300/ [ /s310/ <comment> /s320/ ]
                           /s330/ <statement> /s340/ )*
                     /s350/.

/s280/: out.put(blockopen);
/s290/: out.put(indentreq);
/s300/: out.put(sy,blank);
/s310/: out.put(blockopen);
/s320/: out.put(blockclose,indentreq);
/s330/: out.put(blockopen);
/s340/: out.put(blockclose);
/s350/: out.put(blockclose);
```

Samen met de in hoofdstuk 3 beschreven implementatie van fase 2 kan nu HKL-sourcetekst "geprettyprint" worden. We zullen het voorbeeld daarom besluiten met de verkregen layout van een willekeurig HKL-programma bij twee uitvoerbreedtes te laten zien.

```
MARGIN=35
FASE2 UITVOER:


hklprogram test;
var a,
    # eerste variabele, variabelen
      kunnen alleen integer zijn in
      dit taaltje
    #
```

```
    b, # tweede #
    beta, # ook latinist? #
    c;
    # de rest declareer ik niet #
begin
# hoofd-programma #
if a + b
then begin
     # comment: no serious comment
       yet, though it´s long enough
       with this extension!
     #
     if c + d
        # dit betekent als c+d=0 #
     then begin
          if a * b
             # dit betekent als
               a*b=0, jawel
             #
          then begin
               e := 0; a := b;
               while e + g
                     # zolang e+g>0
                     #
               do begin
                  if g
                  then begin
                       if e
                       then begin
christmas1982 ( MC );
# demonstratie wrap-around, als
  tenminste de margin klein genoeg
  is!
#
if ( ( ( ( ( a + b ) / ( c + d ) )
         + g )
       / exp ( 1 ) )
     + 24 )
then begin
     c := 0;
     if c
     then begin c := 1; e := 0 - g
          end
     else error ( 1 )
     end
                            end
                       end
                  end;
               c := d;
```

```
          # nu volgt een
            aanroep van een
            procedure met 3
            parameters,
            waarvan de tweede
            weer een aanroep
            is van een
            procedure met twee
            parameters, de
            eerste een
            expressie en de
            tweede weer een
            aanroep van nog
            een functie.
            Tevens is in de
            tweede aanroep een
            commentaar
            verwerkt
          #
          print
          ( a * a * a * a,
            functie
            ( ( a + b ) *
              ( c + d ),
              alfa
              ( beta, gamma )
              # belangerijk #
              ),
            0 );
          g := fe; k := henk
          end
     else begin
          x1 := 0;
          x2 := 0
                # ook x2
                  gelijk even
                  nul maken
                #;
          x3 := 0
          end;
     if f + e + g then a := 1;
     allesgehad :=
     nognooitgedaan
     ( eerste_parameter,
       nummer2, par3,
       parameter4,
       tochwelveel, tv, 0, 1 )
     end
end;
c := 0;
```

```
print
( datwashetdan,
  looptochaardig_niet )
end
```

```
MARGIN=72
FASE2 UITVOER:
```

```
hklprogram test;
var a,
    # eerste variabele, variabelen kunnen alleen integer zijn in dit
      taaltje
    #
    b, # tweede #
    beta, # ook latinist? #
    c; # de rest declareer ik niet #
begin
# hoofd-programma #
if a + b
then begin
     # comment: no serious comment yet, though it´s long enough with
       this extension!
     #
     if c + d # dit betekent als c+d=0 #
     then begin
          if a * b # dit betekent als a*b=0, jawel #
          then begin
               e := 0; a := b;
               while e + g # zolang e+g>0 #
               do begin
                  if g
                  then begin
                       if e
                       then begin
                            christmas1982 ( MC );
                            # demonstratie wrap-around, als tenminste de
                              margin klein genoeg is!
                            #
                            if ( ( ( ( ( a + b ) / ( c + d ) ) + g ) /
                                   exp ( 1 ) )
                                 + 24 )
                            then begin
                                 c := 0;
                                 if c
                                 then begin c := 1; e := 0 - g end
                                 else error ( 1 )
                                 end
```

```
                          end
                      end
                  end;
               c := d;
               # nu volgt een aanroep van een procedure met 3
                 parameters, waarvan de tweede weer een aanroep is van
                 een procedure met twee parameters, de eerste een
                 expressie en de tweede weer een aanroep van nog een
                 functie. Tevens is in de tweede aanroep een commentaar
                 verwerkt
               #
               print
               ( a * a * a * a,
                 functie
                 ( ( a + b ) * ( c + d ),
                   alfa ( beta, gamma ) # belangerijk # ),
                 0 );
               g := fe; k := henk
               end
          else begin
               x1 := 0; x2 := 0 # ook x2 gelijk even nul maken #;
               x3 := 0
               end;
          if f + e + g then a := 1;
          allesgehad := nognooitgedaan
                        ( eerste_parameter, nummer2, par3, parameter4,
                          tochwelveel, tv, 0, 1 )
          end
     end;
c := 0; print ( datwashetdan, looptttochaardig_niet )
end
```

6. RESULTATEN, IDEEËN VOOR UITBREIDING EN VERDER ONDERZOEK

Het is gelukt een volledige methode te beschrijven en te realiseren om teksten in willekeurige programmeertalen in twee stappen te prettyprinten, waarbij in de syntaxdefinities door de layoutontwerper, met behulp van een kleine verzameling prettyprint-primitieven, de benodigde layoutaanwijzingen worden ondergebracht.
De realisatie is, vanwege de beperkte tijd voor dit onderzoek, nog niet interactief, hoewel (in het bijzonder met betrekking tot de tweede fase) interactiviteit het uitgangspunt van het gehele ontwerp is, en daarom ook gemakkelijk kan worden bereikt (zie hieronder bij ideeën voor verder onderzoek).

De interactieve opzet vereist een regel voor regel incrementele benadering van het maken van een layout (je wilt zo snel mogelijk uitvoer zien), en een gevolg daarvan is uiteraard dat de layout van een tekstgedeelte niet kan afhangen van een ander gedeelte "verderop". Bijvoorbeeld,

in het geval van een case-statement zal het block behorend bij een case-label altijd op een vaste afstand achter dit label worden geplaatst, en indien er daarna een langer label volgt, wordt het daarbij behorende block ook op een vaste afstand daarachter gezet. Deze blocks zullen dus niet op eenzelfde lijn komen te staan! Bijvoorbeeld:

```
        case getal
        of elf: begin x := 11
                end;

           veertien: begin x := 14
                     end;
```

in plaats van

```
        case getal
        of elf:      begin x := 11
                     end;

           veertien: begin x := 14
                     end;
```

De beschreven methode geeft voor elke programmatekst een layout, die de schrijver van die tekst alleen kan beïnvloeden door de struktuur van zijn programma te veranderen! De layout wordt afgedwongen door de ontwerper van de prettyprint-aanwijzingen in de taalsyntax, en hij dient dus de "prettiest print" te bedenken en te definiëren aan de hand van aanvaardde (zie bijvoorbeeld [4]) of eigen criteria voor tekstlayout. Het voordeel hiervan is layoutstandaardisering van programmatuur in een taal!

Terugkomend op de behandeling van commentaar bij prettyprinten: daartoe moet commentaar een syntactische categorie zijn. Commentaar is daarmee echter nog niet anders op te vatten dan als een verzameling woorden die onderling geen relatie hebben, omdat voor definitie van dergelijke relaties "begrip" van natuurlijke taal benodigd zou zijn! Realistischer is de prettyprinter commentaar te laten uitlijnen aan de hand van in het commentaar geplaatste uitlijnopdrachten (vergelijk het uitlijnen van dit rapport).

We willen besluiten met het geven van ideeën voor uitbreidingen en verder onderzoek.

De verzameling primitieven is naar verscheidenheid voldoende groot, de primitieven zelf echter kunnen nog wel verfijnd worden. Parametrisering van de blank, indentreq en indentstream (indentopen) biedt extra layoutmogelijkheden.
Bij de blank is te specificeren hoeveel spaties er afgedrukt dienen te worden, ingeval hij niet resulteert in een indentering: 0, 1, 2, ... of meer.

De indentreq kan uitgebreid worden met een parameter die aangeeft hoeveel regels er overgeslagen moeten worden, om nog duidelijker scheidingen tussen tekstgedeeltes te kunnen krijgen.
Tenslotte kan de indentstream uitgebreid worden met een parameter die het aantal posities aangeeft dat bij de beginpositie van de indentstream opgeteld moet worden voordat deze positie als indentatie op de indentstack bewaard wordt. Nu is dat aantal immers altijd 0. Juist voor de laatste string uit de indentstream zal dat aantal dan weer van de top van de indentstack afgetrokken moeten worden. Behalve een layout als

```
begin
x := 1
y := 2
z := 3
end
```

is dan bijv. ook mogelijk

```
begin
  x := 1
  y := 2
  z := 3
end.
```

Verder onderzoek is gewenst naar de volgende punten:

- De ontwikkeling van programmatuur die, gegeven een met prettyprint-aanwijzingen uitgebreide syntax, automatisch de benodigde PGEN-invoer genereert (of een PGEN-versie die uitgebreide syntax accepteert) en tevens de uitgebreide syntax controleert op diens tussentekst-syntactische correctheid, zodat alle door die uitgebreide syntax geproduceerde zinnen ook aan de tussentekst-syntax voldoen.

- Het zo mogelijk ontwerpen van een beter notatiesysteem om de regels van de uitgebreide syntax in uit te drukken.

- Het vanuit de bestaande syntax ontwerpen van een uitgebreide syntax voor Pascal of Summer, om de resultaten van het procédé bij een volwaardige programmeertaal te kunnen bekijken.

- Het integreren van fase 1 en fase 2 tot één interactieve prettyprinter. Dit is vrij eenvoudig met PGEN te realiseren door de implementatie van fase 2 en diens globale variabelen, waaronder de benodigde buffers, als globale PGEN-invoer toe te voegen. Vervolgens moeten de semantische acties bij de syntaxregels niet (alleen) primitieven en symbolen wegschrijven naar een (tussen)file, maar moeten ze deze meteen in de tussentekstbuffer zetten. Daarna dient nu bij ieder van die acties ook een aanroep van associate en advanceprint te worden gedaan, om op de tustbuf met het zojuist nieuw toegevoegde token fase 2 "los te laten". De

dan door PGEN gegenereerde parser representeert bij executie, als hij zijn invoer niet uit een file maar van de terminal haalt, de gewenste interactieve prettyprinter.

REFERENTIES

[1] Jan Heering, "Taaldefinities als kern voor een programmeeromgeving", Colloquium Programmeeromgevingen, Mathematisch Centrum, Amsterdam, 22 oktober 1982.

[2] Derek C. Oppen, "Prettyprinting", _ACM Transactions on Programming Languages and Systems_, _2_(1980), 4, pp. 465-483.

[3] Patricia R. Mohilner, "Prettyprinting Pascal programs", _SIGPLAN Notices_, _13_(1978), 7, pp. 34-40.

[4] James L. Peterson, "On the formatting of Pascal programs", _SIGPLAN Notices_, _12_(1977), 12, pp. 83-86.

[5] Paul Klint, _From Spring to Summer_, Proefschrift, Mathematisch Centrum, Amsterdam, 1982.

[6] G. Florijn & G. Rolf, PGEN - A general purpose parser generator, Rapport IW 157/81, Mathematisch Centrum, Amsterdam, 1981.