University of Groningen **Faculty of Mathematics and Physical Sciences**

Department of Computing Science

WORDT
NIET UITGELEEND

NIET
UITLEEN-
BAAR

Een commando shell in een functionele taal

A.R. Holwerda

begeleider: Ir. S. Achterop

18 september 1995

Rijksuniversiteit Groningen
Bibliotheek Informatica / Rekencentrum
Landleven 5
Postbus 800
9700 AV Groningen

RuG

SAMENVATTING

Deze afstudeeropdracht is een onderdeel van een onderzoek dat zich richt op het ontwerp-proces van digitale systemen. Er wordt in dit onderzoek een nieuwe ontwerp-methode voorgesteld waarin het formalisme "functies" centraal staat. Dit betekent dat functies gebruikt worden bij het beschrijven van de systemen en in het ontwerpen van de systemen. Kernwoorden hierbij zijn **"System Semantics"** en **"Transformational Design"**.

Om de beweringen uit het onderzoek te onderbouwen wordt een ontwikkelomgeving gemaakt waarin de ideeën van het onderzoek gebruikt worden. Een onderdeel van deze ontwikkelomgeving is een commandoshell. Deze commandoshell is het onderwerp van deze afstudeeropdracht.

Om in het "functie"-domein te blijven is de commandoshell geschreven in een functionele taal. Tevens is hiervoor gekozen om de nieuwe ontwikkelingen in de functionele talen te onderzoeken.

De opzet van de commandoshell moet zodanig zijn dat uitbreidingen eenvoudig toe te voegen zijn. Dit omdat vele onderdelen later ontwikkeld en toegevoegd gaan worden.

CONCLUSIE

Met de nieuwe ontwikkelingen in de functionele talen kan een groter interactief programma gemaakt worden dat een goede executie snelheid heeft. De commandoshell kan dienen als een voorbeeld hiervan.

Met State Transformers kan op een nette manier I/O gepleegd worden en daarmee is interactie met de gebruiker en het operating systeem mogelijk. De Glasgow Haskell Compiler is een compiler waarmee gecompileerde programma's een goede executie snelheid bereiken.

Het toevoegen van nieuwe onderdelen is door de modulaire opbouw eenvoudig. Een aantal standaard handelingen volstaan.

Hiermee zijn de doelstellingen ruimschoots gehaald.

SAMENVATTING 1

CONCLUSIE 1

1 : INLEIDING 5

1.1 FUNCTIES IN HET ONTWERP-PROCES 5
1.2 DE PROBLEMEN BIJ HET ONTWERPEN VAN SYSTEMEN IN DE PRAKTIJK 5
1.3 SYSTEEM REPRESENTATIE 6
1.4 SYSTEM SEMANTICS. 7
1.5 TRANSFORMATIONAL DESIGN 11
1.6 EIGEN ONDERZOEK : DE COMMANDOSHELL 13

2 FUNCTIONELE PROGRAMMEER TALEN 14

2.1 VOORDELEN FUNCTIONELE PROGRAMMEERTALEN 14
2.2 NADELEN FUNCTIONELE PROGRAMMEERTALEN 15
2.3 STATE TRANSFORMERS 15
2.3.1 Wat is een State Transformer *17*
2.3.2 Variabelen met State Transformers *17*
2.3.3 Sequentiering met State Transformers *19*
2.3.4 I/O met State Transformers *21*
2.3.5 Efficiëntie van de programma's met State Transformers *23*
2.3.6 Conclusie State Transformers *24*
2.4 FUNCTIONELE EN IMPERATIEVE PROGRAMMEER TALEN 24
2.5 FUNCTIONELE EN LOGISCHE PROGRAMMEERTALEN 24

3 DE COMMANDOSHELL 25

3.1 INLEIDING 25
3.2 DE COMMANDO SET 26
3.3 OPBOUW COMMANDOSHELL 28
3.4 HASKELL : GOFER EN DE GLASGOW HASKELL COMPILER (GHC-COMPILER) 30
3.5 TOEVOEGEN VAN EEN NIEUW ONDERDEEL 31

4 DE SYSTEEM SUBSET 33

4.1 INLEIDING 33
4.2 DE SYNTAX VAN DE SYSTEEM SUBSET 33
4.2.1 Lexicale opbouw van de tokens *34*
4.2.2 De grammatica in BNF-notatie *34*
4.2.3 De equivalente grammatica *37*
4.3 VOORBEELDEN VAN ZINNEN UIT DE SYSTEEM SUBSET 38

5 DE DATASTRUCTUUR VOOR DE SYSTEEM SUBSET 40

5.1 DE DATASTRUCTUUR 40
5.2 ACCESS-FUNCTIES VOOR DE DATASTRUCTUUR 41
5.3 VOORBEELDEN VAN OPSLAG IN DE INTERNE DATASTRUCTUUR: 43

6 EEN PARSER EN EEN PRINTER. 45

6.1 DE PARSER VOOR DE SYSTEEM SUBSET 45
6.1.1 De scanner. *45*
6.1.2 De algemene parser functies *46*
6.1.3 De Parser. *50*
6.1.4 De basis type-definties *51*
6.1.5 De atomaire definities *51*

6.2 DE PRETTY PRINTER VOOR DE SYSTEEM SUBSET. 51

6.2.1 Hulp functies en datastructuren voor de Pretty Printers 52
6.2.2 De Pretty Printer voor "programmatekst" 52
6.2.3 De Pretty Printer voor de datastructuur 54

7 DATABASE VAN DE COMMANDOSHELL 55

7.1 DATA CONSTRUCTOREN EN TYPE SYNONIEMEN VOOR DE DATABASE 55
7.2 PROJECTIE FUNCTIES VOOR DE TYPEN SYNONIEMEN VAN DE DATABASE 57
7.3 ALGEMENE BESPREKING VAN DE DATABASE 58
7.4 DATABASE FUNCTIES 59
7.4.1 Inleiding 59
7.4.2 Functies van het database onderdeel Env 63
7.4.3 Functies van het database onderdeel Info 63
7.4.4 Functies van het database onderdeel SysDef 64
7.4.5 Functies van het database onderdeel TupleDef 65
7.4.6 Functies van het database onderdeel BaseDef 67
7.4.7 Functies van het database onderdeel AtomDef 68
7.4.8 Functies van het database onderdeel Commandd 69
7.4.9 Functies van het database onderdeel DefaultCost 70
7.4.10 Functies van het database onderdeel CurSys 71
7.5 DE MODULE COMMHGDB.HS 72

8 DE HELP MODULE 74

9 UTILITIES EN COMPATIBILITEIT 75

9.1 COMPATIBILITEIT 75
9.1.1 Compghc.hs 76
9.1.2. Compgof.hs 77
9.2 SSUTIL.HS 78
9.3 COMMUTIL.HS 79

10 HOOFDPROGRAMMA COMMANDOSHELL 83

10.1 DATA CONSTRUCTOREN VOOR HET HOOFDPROGRAMMA 83
10.2 INLEZEN VAN COMMANDO 83
10.3 HOOFDPROGRAMMA 85

11 DE COMMANDO'S 88

11.1 FILE DESIGN COMMANDO 89
11.2 FILE EDIT COMMANDO 89
11.3 HISTORY COMMANDO 89
11.4 LOAD COMMANDO 89
11.5 PRINT INTERNAL COMMANDO 90
11.6 PRINT TEXT COMMANDO 92
11.7 SHOWSTATE COMMANDO 92
11.8 QUIT COMMANDO 93
11.9 REMOVE COMMANDO 93
11.10 REST COMMANDO'S 93
11.11 SAVE COMMANDO 93
11.12 SET COMMANDO 94
11.13 SHELL COMMANDO 94
11.14 TEMPLATE VOOR HET TRANSFORMATIE COMMANDO 94
11.15 XILINX COMMANDO 94

12 DE BETEKENIS FUNCTIE "XILINX" 95

12.1 FILE VERSCHILLEN 95
12.2 SYNTAX VERSCHILLEN 95
12.3 DATA STRUCTUUR VERSCHILLEN 96
12.4 FOUT AFHANDELING 97
12.5 DE DATABASE EN OPHALEN UIT DE DATABASE 98

12.6 DE CODE 98
12.6.1 Xilidata.hs *99*
12.6.2 Xilieval.hs *99*
12.6.3 Xiligen.hs *99*
12.6.4 Xililam.hs *100*
12.6.5 Xilinorm.hs *101*

13 DE BETEKENIS FUNCTIE "KOSTEN" 102

13.1 DEFINITIE VAN DE KOSTEN FUNCTIE 102
13.2 DE MODULE COMMCOST.HS 103
13.3 DE MODULE COSTMAIN.HS 104

14 TEST PATROON GENERATOR 105

14.1 INLEIDING 105
14.2 DE CODE 106
14.2.1 PatData.hs *107*
14.2.2 Patfunc.hs *109*
14.2.3 Patpars.hs *111*
14.2.4 Patputil.hs *111*
14.2.5 Patstate.hs *113*
14.2.6 Patwutil.hs *115*
14.2.7 Commpatt.hs *116*

LITERATUUR LIJST 118

LIJST MET FIGUREN 119

LIJST MET TABELLEN 120

1 : INLEIDING

1.1 FUNCTIES IN HET ONTWERP-PROCES

Deze afstudeeropdracht is een onderdeel van een onderzoek dat zich richt op het proces van het ontwerpen van digitale systemen. De manier waarop nu systemen ontworpen worden kent een aantal nadelen. Door in het ontwerp-proces vergaand met het concept "functies" te gaan werken kunnen de nadelen voor een deel opgelost worden. In het onderzoek worden functies gebruikt bij :

- het beschrijven van systemen.
- het redeneren over / rekenen aan systemen.
- het toekennen van betekenissen aan beschrijvingen van systemen.

De voordelen van functies zijn :

- Functies zijn "*referential transparent*". Dit wil zeggen dat iedere expressie steeds dezelfde waarde heeft. Nog anders gezegd, er zijn geen variabelen. Dit heeft grote voordelen bij het "rekenen" aan een beschrijving van een systeem omdat geen rekening gehouden hoeft te worden met het mogelijk veranderen van een waarde van een expressie.
- Functies maken een wiskundige bewijsvoering mogelijk
- Functies zijn declaratief. Ze vertellen wat er gebeuren moet en niet hoe het gebeuren moet. Vertellen wat er gebeuren moet is eenvoudiger dan vertellen hoe het gebeuren moet.
- Functies maken veel aspecten expliciet.

De inleiding op het gebruik van functies wordt begonnen in de nu volgende paragraaf. Hierin wordt een denkbeeldige schets gemaakt van de manier waarop nu ontworpen wordt. Hieruit komen de nadelen van de huidige werkwijze duidelijk naar voren.

1.2 DE PROBLEMEN BIJ HET ONTWERPEN VAN SYSTEMEN IN DE PRAKTIJK

Stel dat de firma X in 1970 begonnen is met alleen de huidige directeur. Als eerste produkt werd de schakeling X-1 gemaakt. Bij het produceren van deze schakeling deed de directeur alles alleen, onder andere het ontwerpen, de in- en verkoop en het testen. Produkt X-1 werd een groot succes en het vervolg, de X-2, liet niet lang op zich wachten.

Het werk aan de X-2 kon niet meer alleen gedaan worden en er werden een aantal medewerkers aangetrokken. Ieder van deze medewerkers ging zich richten op een bepaald deel van het te produceren systeem. Door een constant succes groeide de firma X en in 1995 kent de firma een groot aantal werknemers en een groot aantal afdelingen. De schakelingen die gemaakt worden zijn in de loop van de tijd gegroeid, zowel in omvang en als in complexiteit.

In 1994 besluit de directeur van de firma, dat een nieuw systeem, de X-95, gemaakt moet worden. De specificatie van het systeem komt in de vorm van een lijvige functionele beschrijving. De ontwerp-afdeling zet zich aan het werk en ontwerpt het systeem. De testafdeling gaat het nieuw ontworpen systeem na verloop van tijd testen. Maar hier ontstaat een probleem. De ontwerp-afdeling heeft het systeem afgebeeld als een schema. Maar de testafdeling kan hier niets mee. Die werkt met timingsdiagrammen voor het testen van het systeem. En daarom maakt deze afdeling een eigen versie van het systeem, uitgedrukt in timingsdiagrammen. Als gevolg hiervan bestaan er meerdere versie van het systeem. De directeur kent systeem X-95. De ontwerp-afdeling kent systeem X-95', de testafdeling X-95'' enzovoort. En dit is vreemd want iedere afdeling praat over hetzelfde systeem.

De kern van de zaak is dat, door het groeien van de te maken systemen, het ontwikkelen ervan wordt opgedeeld in stukken. Tussen de verschillende afdelingen is het nodig dat er informatie wordt uitgewisseld over het systeem. En dit is moeilijk omdat in ieder fase gebruik gemaakt wordt van een andere representatie van het systeem. Gevolg hiervan is dat in iedere afdeling zijn eigen systeem maakt . En er moet maar gehoopt worden dat deze systemen qua gedrag hetzelfde zijn.

Een ander probleem dat is ontstaan in de loop van de tijd ligt in ontwerpen van de systemen. In 1970 was het systeem klein en kon de directeur het alleen af. De systemen werden echter steeds groter en complexer.

Voor het ontwerpen heeft dit onder andere de volgende twee gevolgen :

- het bewijzen van de correctheid van het uiteindelijke resultaat wordt moeilijker.
- de ontwerptijd neemt toe.

Vooral het eerste punt is van vitaal belang. Als niet aangetoond kan worden dat het systeem doet wat het doen moet, dan is het resultaat op zijn zachts gezegd dubieus. Een oplossing voor beide problemen zou kunnen zijn het automatiseren van het ontwerpen.

Deze praktijkschets is natuurlijk niet compleet. Maar ze geeft wel een beeld van de problemen die kunnen ontstaan bij het maken van systemen. Dat in de praktijk toch systemen op deze manier gemaakt worden kan daarom op meerdere manieren worden uitgelegd. Een daarvan is dat alles wel wat meevalt. Een andere is dat we tot nu toe geluk hebben gehad. Duidelijk is in elk geval dat in het ontwerp-proces het een en ander verbeterd kan worden.

Samengevat zijn de problemen uit de praktijk :

- informatie uitwisseling is moeilijk zodat er meerdere versie van hetzelfde systeem gemaakt worden.
- het ontwerp-proces duurt te lang en de correctheid ervan is moeilijk te bewijzen.

Voor een oplossing van het eerste probleem kan de System Semantics uit § 1.4 een uitkomst bieden. Voor een oplossing van het tweede probleem kan de theorie van de Transformational Design uit § 1.5 ingezet worden. Na de informele manier van deze paragraaf worden in de volgende paragrafen op een meer formele manier de verschillende aspecten van het ontwerp-proces besproken.

1.3 SYSTEEM REPRESENTATIE

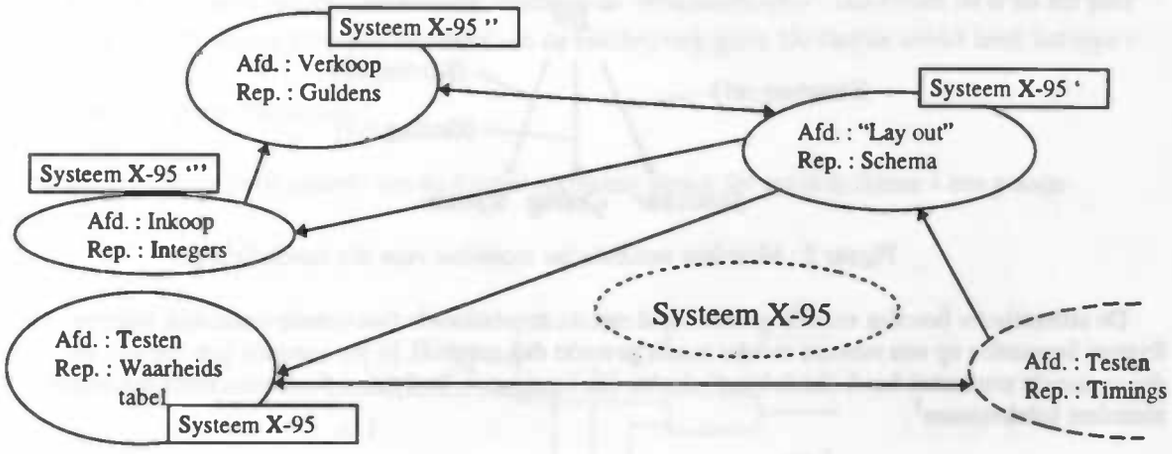
Een digitaal systeem kan op een aantal manieren gerepresenteerd worden, bijvoorbeeld :

- een specificatie : een functionele beschrijving van wat het systeem moet doen.
- een schema : een grafische weergave van het systeem
- een onderdelen lijst : een lijst van onderdelen
- een prijs : een eenvoudige maar zeer belangrijke representatie
- een aantal timingsdiagrammen : hoe het systeem reageert op bepaalde invoer
- een netlist : een veel gebruikte manier om een systeem te beschrijven. Een netlist bestaat uit een lijst van onderdelen en verbindingen tussen de verschillende pinnen.
- .....

Met het maken van een systeem zijn in de praktijk meerdere afdelingen bezig. Iedere afdeling gebruikt een representatie van het te maken systeem die het beste past bij de aspecten die door hen bekeken moeten worden. Dit houdt in dat voor een en hetzelfde systeem een aantal verschillende representaties worden gebruikt.

Als iedere afdeling zonder interactie met andere afdelingen aan het systeem zou kunnen werken, dan waren er geen problemen met betrekking tot die verschillende representaties van het systeem. Iedere afdeling leeft dan als het ware op een eiland en werkt daarop aan het systeem. Maar in de praktijk is die interactie wel nodig. De resultaten van de ene afdeling zijn de invoer voor een andere afdeling, en omgekeerd. Een gevolg van het overgaan van de ene afdeling naar de andere is dus ook een overgang van representatie.

En dit leidt tot problemen omdat het moeilijk, of soms zelfs niet mogelijk is, om de ene representatie om te zetten in een andere representatie. Onder andere omdat veel representaties niet volledig expliciet zijn zodat bepaalde informatie niet beschikbaar is bij de transformatie. In figuur 1 wordt deze praktijkschets uitgebeeld. Iedere afdeling heeft zijn eigen representatie en via de pijlen wordt, denkbeeldig, informatie uitgewisseld. En in het midden staat het systeem waar het allemaal om draait en waarvan vele afzonderlijke versies voorkomen.



Figuur 1 : De ontwerp-praktijk met meerdere "gelijke" systemen.
Index : Afd. = Afdeling
Rep. = Representatie van het systeem.

Het verschil in gebruikte representatie kan ook op een andere manier beschreven worden. Iedere afdeling werkt aan hetzelfde systeem maar geeft aan dit systeem een andere betekenis. De afdeling inkoop geeft het systeem de betekenis : "Boodschappenlijst". De afdeling testen geeft het systeem de betekenis : "Gedrag". Met deze benadering wordt gewerkt in de theorie van de System Semantics die in § 1.4 besproken wordt.

1.4 SYSTEM SEMANTICS.

System Semantics is het beschrijven van systeem eigenschappen door middel van semantische functies. Een systeem is hierin een verzameling componenten die onderling verbonden zijn. Het aantal componenten en de verbindingen er tussen veranderen niet.

Om systemen te kunnen beschrijven is een taal nodig. Noem deze taal $\mathcal{S}$[1] en laat χ een element zijn uit deze taal. Dan is χ een beschrijving van een systeem en wordt χ een **formal function definition** genoemd. Belangrijk is dat χ slechts een syntactische beschrijving van het systeem is wat ondermeer inhoudt dat er geen betekenis aan mag worden toegekend.

Een semantiek (een eigenschap) van een systeem wordt expliciet geïntroduceerd door middel van een model, zeg $\mathcal{M}$. Een model $\mathcal{M}$ is een paar $(\mathcal{D},m)$ waarin $\mathcal{D}$ een bepaald betekenisdomein is en m een semantische functie die een beschrijving van het systeem afbeeld op dit betekenisdomein. De verzameling van betekenisdomeinen voor het onderzoek zijn bijvoorbeeld :

$$\mathcal{E} = \{ \mathit{Gedrag}, \mathit{Kosten}, \mathit{Structuur}, \mathit{Onderdelenlijst}, \{0,1\}, \ldots \}$$

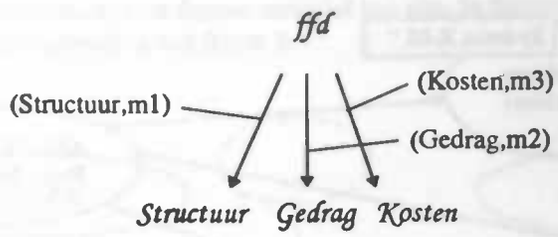
De functie m is te noteren als :

$$m \in \mathcal{S} \rightarrow \mathcal{D} \text{ met } \mathcal{D} \in \mathcal{E}$$

Met verschillende modellen is het mogelijk om verschillende semantieken, betekenissen, aan een beschrijving toe te voegen. Dit is in figuur 2 uitgebeeld.

---

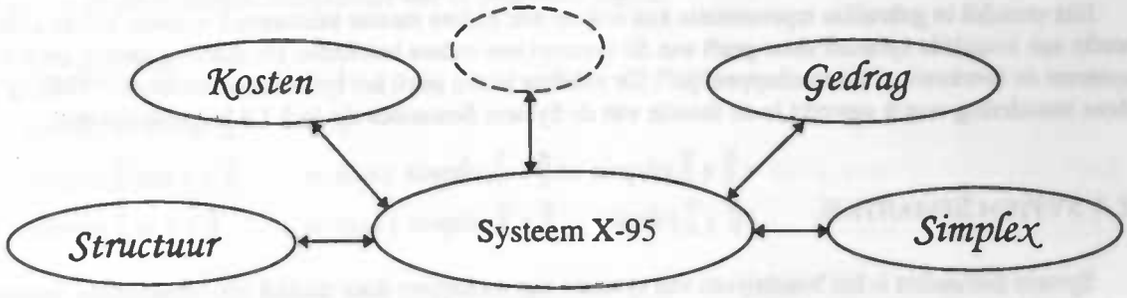
[1] In het onderzoek wordt hiervoor de taal van de Systeem Subset genomen. In hoofdstuk 4 wordt deze taal besproken.



Figuur 2 : Meerdere semantische modellen voor één beschrijving.

De semantische functies worden gedefinieerd met de denotationele functionele semantiek waarmee in System Semantics op een ruimere manier wordt gewerkt dan normaal. In het normale gebruik van de denotationele semantiek heeft een "object" slechts één betekenis[2]. In System Semantics heeft een object meerdere betekenissen[3].

Met System Semantics kan figuur 1 worden omgezet in figuur 3. Het systeem staat hierin centraal en de verschillende betekenis domeinen éromheen. Met de pijlen worden de semantische functies weergeven.



Figuur 3 : Het systeem centraal.

Als voorbeeld van System Semantics zal nu een systeembeschrijving en een aantal semantische functies worden gegeven. Neem als een systeembeschrijving het onderstaande systeem *Sel* uitgedrukt in de taal van de Systeem Subset. Deze beschrijving is een element van de taal *S*.

```
Sel ∈ (B,B) -> (B,B,B,B)
Sel (i1,i2) = (u1,u2,u3,u4)
      where u1    = and2 ((ni1),(ni2))    ;
            u2    = and2 ((ni1),i2)       ;
            u3    = and2 (i1,(ni2))       ;
            u4    = and2 (i1,i2)          ;
            ni1   = nand1 i1              ;
            ni2   = nand1 i2
      endwhere
```

Iedere kenner van digitale systemen herkent hierin een 2-naar-4 bits selector. Maar er mag geen betekenis aan deze beschrijving worden toegekend. Dit wordt gedaan met de semantische functies.

---

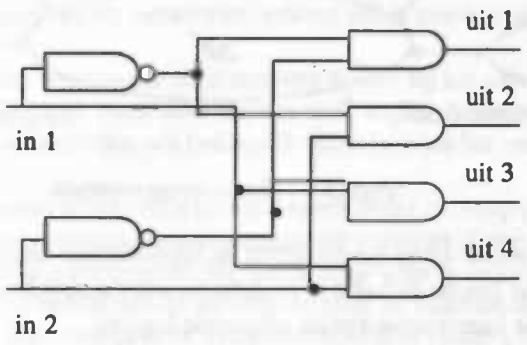
[2] Een voorbeeld hiervan is een programmatekst. Deze tekst heeft één betekenis.

[3] Wordt mij gevraagd wat het bijzondere aan System Semantics is dan vind ik dit aspect het meest opvallend. Het gehele idee van System Semantics is achteraf gezien niet meer dan logisch. Maar er moet maar iemand op komen.

Een bekend voorbeeld van een semantische functie is de "structuurfunctie". Het model $\mathcal{M}$ is nu het paar (*"Structuur", struct*) waarin *struct* de structuur van de beschrijving geeft. De functie *struct* heeft het type :

$$struct \in \mathcal{S} \rightarrow \text{"Structuur"}$$

Als *representatie* van de *waarde* van de functie-applicatie *struct Sel* wordt in figuur 4 een plaatje opgeleverd.



Figuur 4 : Representatie van de waarde van de functie *struct Sel*

Er is een onderscheid gemaakt tussen de waarde en de representatie van de waarde. De representatie in het voorbeeld is een tekening. Maar het kan net zo goed een postscript file zijn. In de nu volgende voorbeelden wordt dit onderscheid weggelaten en wordt direct met een representatie gewerkt. Het is wel belangrijk om te bedenken dat dit onderscheid er is.

Een ander voorbeeld van een semantisch model is het paar (*{0,1}, simplex*). De semantische functie *simplex*[4] bepaalt het combinatorische gedrag van een systeembeschrijving,. Het type van de functie simplex is :

$$simplex \in \mathcal{S} \rightarrow \{0,1\}$$

Het resultaat van de functie-applicatie *simplex Sel* staat in figuur 5.

i2	i1	u1	u2	u3	u4
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Figuur 5 : Het resultaat van de applicatie van *simplex Sel*

Met het model (*Kosten, cost*) kan aan een systeembeschrijving de betekenis "wat kost het ?" gegeven worden. Het type van de functie *cost* is :

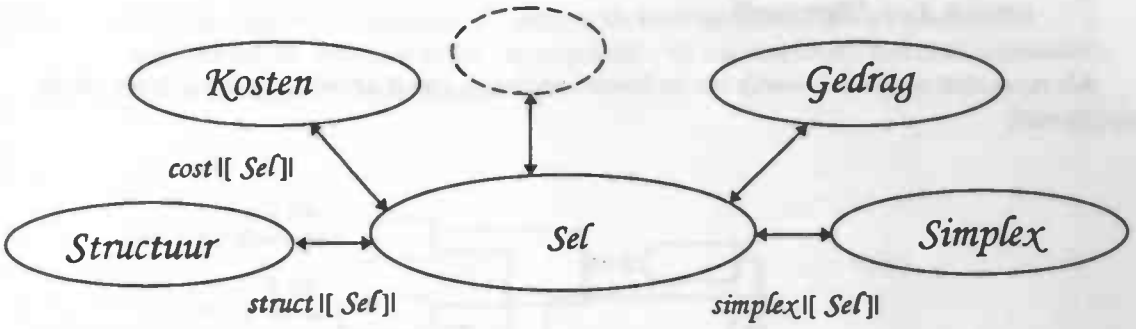
$$cost \in \mathcal{S} \rightarrow Kosten$$

Voor het systeem *Sel* is het resultaat van de functie-applicatie *cost Sel* :

$$4 * cost\ and2 + 2 * cost\ nand1$$

[4] De naam simplex wordt in deze context gebruikt voor het beschrijven van combinatorische systemen. Dit zijn systemen zonder tijdsafhankelijkheden.

Uit dit voorbeeld blijkt dat een semantische functie recursief kan zijn. In figuur 6 wordt dit voorbeeld uitgebeeld. Deze figuur is een uitbreiding van figuur 3.



Figuur 6 : Het systeem centraal.

Voor de definitie van een semantische functie wordt de denotationele functionele semantiek gebruikt. Een voorbeeld van een semantische functie definitie is de definitie van de *simplex* functie. In deze definitie wordt uitgegaan van de gebruikelijke interpretatie van de propositie logica.

$$
\begin{aligned}
&simplex[\![\ \text{"True"}\ ]\!] &&= 1\\
&simplex[\![\ \text{"False"}\ ]\!] &&= 0\\
&simplex[\![\ \text{and x y}\ ]\!] &&= \min\,(\ simplex[\![\ \text{x}\ ]\!]\,)\,(\ simplex[\![\ \text{y}\ ]\!]\,)\\
&simplex[\![\ \text{or x y}\ ]\!] &&= \max\,(\ simplex[\![\ \text{x}\ ]\!]\,)\,(\ simplex[\![\ \text{y}\ ]\!]\,)\\
&\ldots\ldots
\end{aligned}
$$

In hoofdstuk 13 wordt een kosten functie op deze manier gedefinieerd.

Deze bespreking van System Semantics is beperkt en toegespitst op het gebruik in het onderzoek. Een volledige bespreking is te vinden in [5].

In het onderzoek wordt een beperkte vorm van System Semantics gebruikt. In de theorie van System Semantics worden de systemen beschreven met relaties. In het onderzoek worden de systemen beschreven met functies. Dit is een beperking omdat in systemen relaties kunnen voorkomen. Voorbeelden hiervan zijn :

- een weerstand.
- tristate.
- open collector

Dit zijn componenten die niet eenduidig een invoer- en uitvoerkant kennen en daarom niet met functies kunnen worden beschreven. Door in het onderzoek te kiezen voor functies worden de mogelijk te beschrijven systemen beperkt. Toch is gekozen voor functies omdat :

- veel van de te bestuderen systemen wel met functies beschreven kunnen worden
- de implementatie van een functionele taal sneller is.

Dit laatste argument hangt samen met het evalueren van systemen. Relaties kunnen worden geëvalueerd met een logische taal, functies met een functionele taal. Functionele talen zijn op dit moment echter veel sneller dan logische talen en hebben daarom de voorkeur.

De beperking tot functies is overigens niet erg beperkend. Omdat functies een speciaal geval zijn van relaties (en daarom ook sneller geëvalueerd kunnen worden door een functionele taal) kan later altijd nog worden overgegaan op het gebruik van relaties en een logische taal, wanneer de ontwikkelingen van logische talen zover zijn, dat op een redelijke snelheid een relatie geëvalueerd kan worden.

In deze afstudeeropdracht zijn de raakvlakken met System Semantics :

- De commandoshell is een onderdeel van een ontwikkelomgeving die werkt met System Semantics.
- Er zijn twee semantische functies gemaakt. Een kosten functie en een functie die een systeem beschrijving in een Xilinx formaat omzet. Deze laatste functie kan gebruikt worden om een systeem te testen met het Xilinx pakket.

- Er zijn twee semantische functies gemaakt. Een kosten functie en een functie die een systeem beschrijving in een Xilinx formaat omzet. Deze laatste functie kan gebruikt worden om een systeem te testen met het Xilinx pakket.

1.5 TRANSFORMATIONAL DESIGN

In § 1.2 zijn de problemen rond het ontwerpen van een systeem op informele wijze bekeken. Doordat de systemen steeds complexer en groter worden ontstaan de volgende problemen :

- Het ontwerpen kost steeds meer tijd.
- Het bewijzen van de correctheid van het ontworpen systeem wordt steeds moeilijker.

Wanneer tools gemaakt kunnen worden die ondersteuning bieden bij het ontwerpen kunnen de beide problemen gedeeltelijk worden opgelost. Deze tools moeten zelf[5] (deel)-ontwerp-stappen kunnen doen zodanig dat het systeem correct blijft. Deze tools zijn een belangrijk onderdeel van het omvattende onderzoek.

Om goede tools te kunnen maken is het nodig om de stappen in het ontwerp-proces expliciet te maken[6] . In het huidige ontwerp-proces zijn de ontwerp-stappen meestal niet expliciet of niet expliciet genoeg. Door deze vaagheden zijn de huidige tools niet goed opgezet. De architectuur waarop de tools gebouwd worden is niet goed. Het echte werk, het expliciet maken van de ontwerp-stappen en daarmee het maken van een goede structuur, blijft liggen omdat dit het moeilijkste is. Is dit moeilijke deel echter geklaard, dan zijn de overige problemen (relatief) eenvoudig. Door te gaan ontwerpen met Transformational Design wordt men gedwongen de ontwerp-stappen expliciet te maken.

In het kader van het onderzoek wordt ontwerpen gedefinieerd als :

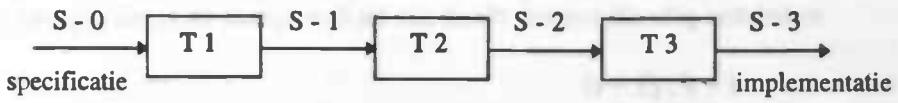
> ***"Ontwerpen is het omvormen, transformeren, van de specificatie in een implementatie van deze specificatie"***

Hierbij is een beschrijving van een systeem een specificatie als de beschrijving geïnterpreteerd kan worden als een beschrijving van het probleem. Een beschrijving van een systeem is een implementatie als de beschrijving geïnterpreteerd kan worden als de beschrijving van een "fysiek ding". Een "fysiek ding" gemaakt van een implementatie wordt een realisatie genoemd. Het valt op dat zowel specificatie en implementatie een beschrijving van het systeem zijn en dat het verschil gemaakt wordt door de keuze van interpretatie.

Transformational Design is het omvormen van een beschrijving van een systeem in een andere beschrijving van het systeem met behulp van transformatie regels. Met als doel om uiteindelijk een implementatie van het systeem te krijgen. Deze transformatie regels moeten ***"correctness-preserving"*** zijn. Dit wil zeggen dat na toepassen van een regel het gedrag van het systeem gelijk blijft.

Een implementatie is ***"correct by construction"*** als iedere ontwerp-stap ***"correctness-preserving"*** is. Als bewezen kan worden dat iedere stap "correctness-preserving" is, dan is daarmee ook bewezen dat het ontwerp correct is. Hiermee is het tweede probleem opgelost.

De transformatie regels zijn in hun gedrag functies. Een beschrijving van een systeem is het argument en een gewijzigde beschrijving wordt opgeleverd. Door het functieaspect is het noodzakelijk om alles expliciet op te schrijven. Hierdoor is het mogelijk geworden om tools te maken. In figuur 7 is het ontwerpen met Transformational Design uitgebeeld.



Figuur 7 : Voorbeeld van het ontwerpen met Transformational Design.
S -1 en S -2 zijn tussentijdse beschrijvingen van het systeem.

---

[5] Dit is nog toekomstmuziek. Eerst zal de ontwerper nog steeds mee moeten helpen door sturing te geven etc. Maar het uiteindelijke doel is dat de tools steeds meer zelf gaan doen. Hiermee is de ontwerptijd terug te brengen.

[6] Een van de belangrijke stellingen uit het onderzoek.

Als T 1 en T 2 correctness preserving zijn is het ontwerp S - 3 correct by construction. Veel van de transformatie regels uit Transformational Design worden opgebouwd uit een aantal basisregels. Deze basisregels worden nu besproken.

- Folding : Bij deze regel wordt een expressie herkent als een functieaanroep met zekere argumenten. De expressie wordt vervangen door de functieaanroep met deze argumenten. Laat e de volgende expressie zijn :

  $$e = exp\ (4)$$

  en laat f de functie

  $$f\ x = exp\ x$$ zijn.

  Dan wordt met folding de expressie e omgezet in $f\ 4$. Dat dit klopt is direct duidelijk vanwege de substitutie van gelijken door gelijken.
- Unfolding : Deze regel is het omgekeerde van folding en is het uitrekenen van een functie . Laat e de volgende expressie zijn :

  $$e = exp\ (z)$$

  en laat f de functie

  $$f\ x = exp\ x$$ zijn.

  Dan wordt met unfolding de functieaanroep $f\ z$ omgezet in de expressie e. Dat dit klopt is direct duidelijk vanwege de substitutie van gelijken door gelijken.
- Generalisatie : Deze regel wordt toegepast in het geval van een aantal ongeveer gelijke expressies. Er wordt een nieuwe, algemenere, expressie gemaakt waarmee de oorspronkelijke expressies, door het kiezen van een goede parameter, uitgedrukt kunnen worden. Neem aan dat de volgende functie gegeven is :

  $$f\ x\ y = (x + y)\ .\ 4$$

  Een nieuwe functie kan dan zijn :

  $$h\ x\ y\ z = (x + y)\ .\ z$$

  zodat de f uitgedrukt in h kan worden :

  $$f\ x\ y = h\ x\ y\ 4 \qquad h\ x\ y\ z = (x + y)\ .\ z$$
- Eureka : Met deze basisregel wordt een geheel nieuwe functie gevonden. Dit is een creatief proces, waarvoor misschien de hulp van de ontwerper nodig is. Of misschien kunnen hier AI-technieken gebruikt worden. Neem aan dat de volgende expressie gegeven is :

  $$a + b\ .\ (3 + c)$$

  Een nieuwe functie kan dan zijn :

  $$f\ x\ y\ z = x + y\ .\ (3 + z)$$

  Hier is zelf een nieuwe functie "*f*" ontdekt, *eureka!!!*.

Transformational Design is hiermee kort besproken. Behandeld is hoe Transformational Design gebruikt kan worden bij het ontwerpen en er zijn enige basis transformatie regels gegeven. Een bespreking van Transformational Design is te vinden in [14]

Een raakvlak met Transformational Design in deze afstudeeropdracht, is dat de commandoshell gebruikt gaat worden ontwikkelomgeving die werkt met Transformational Design.

Een ander belangrijk raakvlak is gelegen in de bewijsbaarheid van de transformatie regels. Er moet bewezen worden dat de regels ***correctness preserving*** zijn. Het bewijzen van deze eigenschap moet gedaan worden door een bewijs te leveren over de implementatie.

In het onderzoek worden deze regels geïmplementeerd met functies door een functionele taal te gebruiken[7]. Het leveren van een bewijs over een functie is in het algemeen beter te geven. De kern van de betere bewijsbaarheid van functies is gelegen in de ***"referential transparency"***. Zoals eerder gezegd betekent dit dat iedere (deel)-expressie overal dezelfde waarde heeft. Hierdoor hoeft geen rekening gehouden te worden met een mogelijk veranderen van de waarde van een expressie.

1.6 EIGEN ONDERZOEK : DE COMMANDOSHELL

Om de beweringen uit het onderzoek te onderbouwen wordt een ontwikkelomgeving gemaakt waarin de ideeën van het onderzoek gebruikt worden. De ideeën zijn :

- Het gebruik van functies als belangrijkste formalisme in hét ontwerp-proces
- Het gebruik van System Semantics voor het toekennen van betekenis en representatie aan een beschrijving van het systeem
- Het gebruik van Transformational Design voor het ontwerpen van een systeem

Een onderdeel van deze ontwikkelomgeving is de commandoshell. Deze shell is te vergelijken met een IDE[8]. De shell zorgt onder andere voor de interactie met het besturingssysteem en de gegevens van de ontwikkelomgeving.

Deze commandoshell is voor het grootste deel onderwerp van deze afstudeeropdracht en zal in de rest van dit verslag besproken worden. In hoofdstuk 3 wordt een inleiding gegeven over de commandoshell.

---

[7] Dit is het eigenlijke raakvlak. De commando-shell is functioneel en de transformatie regels zijn functioneel.

[8] Integrated Development Environment

2 FUNCTIONELE PROGRAMMEER TALEN

Voor het concreet maken van de in het onderzoek voorgestelde aanpak voor het ontwerpen van systemen wordt een ontwikkelomgeving gemaakt. Deze wordt geprogrammeerd in de functionele programmeertaal Haskell. De redenen hiervoor zijn :

- functies zijn het belangrijkste formalisme in het onderzoek. Een functionele programmeertaal past hier dan ook goed bij.
- functioneel programmeren kent belangrijke voordelen. Zie § 2.1
- onderzoeken van de nieuwste ontwikkelingen in functionele programmeer talen. Met deze ontwikkelingen kunnen de nadelen (gedeeltelijk) worden opgelost. Zie § 2.2 en § 2.3

2.1 VOORDELEN FUNCTIONELE PROGRAMMEERTALEN

De voordelen van functionele talen zijn grotendeels gelijk aan de voordelen van de functies uit § 1.1. Vreemd is dit niet omdat een functionele programmeertaal een evaluator van functies is. De voordelen worden nog een keer opgenoemd, nu vanuit het oogpunt van programmeren. De vergelijking wordt gemaakt met een imperatieve programmeertaal. De voordelen zijn dan :

- Functionele programma's vertellen "wat" er moet gebeuren, niet hoe het moet gebeuren. Dit heeft de volgende voordelen :
  - Programma's kunnen korter zijn. Het is gemakkelijker te vertellen ***wat*** er gebeuren moet dan ***hoe*** het gebeuren moet.
  - Functionele programma's zijn ***abstracter*** en liggen dichter tegen een ***specificatie*** aan.
  - Functionele programma's bieden betere mogelijkheden tot parallellisme omdat de compiler een veel grotere vrijheid heeft in het bepalen hoe een expressie wordt geëvalueerd. Dit kan worden uitgebuit bij verdelen van evaluaties over meerdere processors.
- Functionele talen kennen geen state. Dit heeft het volgende voordeel :
  - Het programma is ***referential transparent***[9] . Iedere expressie heeft overal dezelfde waarde.
    Een gevolg hiervan is dat functioneel programma geen variabelen kennen. Hierdoor is een programma beter te begrijpen omdat helemaal geen rekening gehouden hoeft te worden met het mogelijk veranderen van een waarde. Dit in tegenstelling tot bijvoorbeeld imperatieve talen.
- Bewijzen over het programma zijn makkelijker te maken. Dit komt door de volgende punten :
  - Functionele programma's liggen dicht bij de wiskunde. Dus bewijsvoering, leesbaarheid e.d. zijn beter en intuïtiever.
  - Functionele talen kennen geen expliciete sequencering. In een imperatieve taal is door de expliciete sequencering, de ";" uit Pascal bijvoorbeeld, de bewijsvoering zeer moeilijk want het gehele programma moet in ogenschouw genomen worden (bijvoorbeeld door middel van post- en pre-condities). Anders gezegd : imperatieve talen kennen state en zijn daardoor moeilijk in het bewijzen van eigenschappen.
  - Functionele programma's zijn ***referential transparent*** (zie vorige punt) en maken daardoor de bewijsvoering eenvoudiger.
- Functionele programma's kennen hogere orde functies. Dit zijn functies die functies als argumenten hebben of functies die functies als resultaat opleveren. Functies zijn first-class citizen. Hiermee is het mogelijk om zelf controle structuren te maken. Zo kennen functionele talen geen for lus. Maar die kun je zelf maken.
- Haskell kent een sterk typesysteem waardoor meer programmeer fouten worden opgespoord. Een onderscheid tussen imperatieve programma's en functionele programma's is het volgende :
  "Een geldig functioneel programma is in zijn geheel een grote expressie. Wordt daaruit een deel weggehaald dan resulteert dit direct in een foutmelding bij het compileren. Een imperatief programma bestaat uit een opeenvolging van statements. Wordt uit een geldig programma een statement uitgehaald, dan hoeft dit nog niet direct tot een fout te leiden".

---

[9] Dit wil zeggen dat iedere expressie overal dezelfde waarde heeft. Hierdoor kan, bijvoorbeeld, zonder nadenken een expressie worden vervangen door zijn waarde. En dit in het gehele programma.

- Haskell kent lazy-evaluation. Dit wil zeggen dat een expressie pas geëvalueerd wordt als de waarde van de expressie nodig is. Hiermee zijn bijvoorbeeld op eenvoudige manier oneindige datastructuren mogelijk. En dit wordt in het programma ook gebruikt. Zo kan de database eenvoudig worden uitgebreid.

2.2 NADELEN FUNCTIONELE PROGRAMMEERTALEN

Maar waarom worden functionele talen dan niet veel meer gebruikt. Dit komt omdat functionele talen ook nadelen hebben. De nadelen zijn :

- Programma's geschreven in functionele talen zijn traag in de uitvoering en vragen veel geheugen. Door nieuwe compiler-technieken worden deze nadelen steeds verder teruggedrongen.
- Functionele talen kennen geen state. Een van de gevolgen hiervan is dat I/O niet mogelijk is. Dit omdat I/O bij uitstek een actie is waarin state een rol speelt, nl. de toestand van de buitenwereld, bijvoorbeeld :
  - Is een toets op het toetsenbord ingedrukt?
  - Is een file geopend?
  - Wat is het huidige record in de file.
  - .....

Omdat I/O soms toch wel nodig is[10], is dit in de meeste huidige functionele programmeertalen toegevoegd, maar niet op een fraaie wijze. Onder andere zijn in sommige talen bepaalde delen van het programma niet meer volledig functioneel. Met State Transformers is state, en daarmee I/O, wel op een fundamenteel, nette manier mogelijk. Deze State Transformers worden nu besproken.

2.3 STATE TRANSFORMERS

In de geschiedenis van functioneel programmeren werd een state als een expliciete parameter meegegeven [2]. Dit is niet zo verwonderlijk want state is eigenlijk niets anders dan een functie van de tijd[11] naar een resultaat. Neem bijvoorbeeld het imperatieve programma :

```
a := 0
while true do
    a = a + 2
end do
```

Dan is de state voor de variabele a :

0	1	2	3	4	5	6	7	8	...
0	2	4	6	8	10	12	14	16	18

waarbij de nummers uit de bovenste rij het aantal executie van de while-lus zijn[12].

Een functioneel programma voor deze herhaling kan zijn :

```
type Result = Int

funstart :: Result
funstart = snd (two_increment(0,0))
```

---

[10] "Soms" is in deze zin eigenlijk "altijd". Een programma zonder I/O is redelijk bijzonder. In strikte zin gesproken is zo'n programma zelfs nutteloos, want ook het presenteren van een eindresultaat van is I/O. Maar deze vorm van I/O wordt niet tot "I/O in een functioneel programma" gerekend.

[11] Aangenomen dat de toestand bepaald wordt in de tijd.

[12] Aangenomen dat iedere executie van de while-lus samenvalt met een tijdstap.

```
two_increment :: (Int,Result) -> (Int,Result)
two_increment (x,y) = two_increment (x+1,y+2)
```

De state is hier het eerste element van het paar, het resultaat het tweede element. Nu geeft deze functie dezelfde lijst als boven. Dit is grafisch voor te stellen als in figuur 8 :

Figuur 8 : Een grafische weergave van state.

Dus de bewering dat met functionele talen geen state mogelijk is, is niet waar. De state moet wel zelf expliciet gemaakt worden. En dit blijkt in de praktijk lastig, maar het kan. Een mogelijk type in Haskell voor deze state kan zijn :

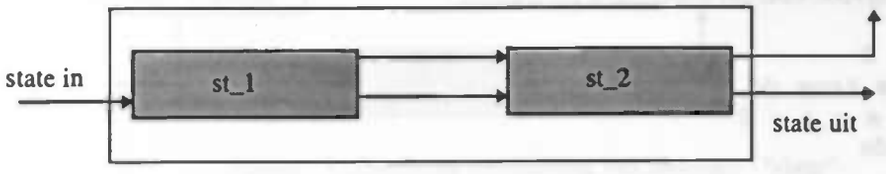
```
ST s r
```

met s het type van de state en r het type van het resultaat.

Wat nu ook nog gewenst is, zijn functies om dit soort berekeningen samen te stellen. Dit is vergelijkbaar met de ";" uit Pascal. Noem deze functie : *thenST*. Het type van deze functie zou moeten zijn :

```
thenST :: (ST s r) -> ( r -> ST s b) -> ST s b
```

Dit is een functie die een state neemt, op het resultaat van deze state een functie loslaat, en een nieuwe state en resultaat oplevert. Dit is grafisch voor te stellen als in figuur 9 :



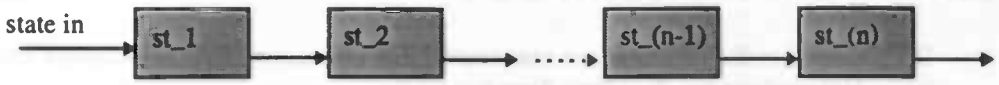
Figuur 9 : Grafische voorstelling van de functie *thenST*.

In de recente ontwikkeling is een ADT gevonden waarmee het bovenstaande mogelijk is [7]. Bij deze ADT zijn altijd twee functies aanwezig :

- `returnST :: a -> ST s a`
- `thenST :: (ST s r) -> ( r -> ST s b) -> ST s b`

Met deze twee functies is het mogelijk om berekeningen samen te stellen. Verder is aan deze ADT belangrijk dat het een monad is. Dit houdt in dat sequentiering optreedt in een samengestelde berekening. Dit betekent dat in de samenstelling uit figuur 10 iedere `st_i` slechts een keer gebruikt gaat worden. Het gevolg hiervan is, dat het mogelijk is om met één `s` te gaan werken die met in-place updates actueel gehouden wordt. En dit is de state zoals wij die kennen uit imperatief programmeren. En die wij soms graag willen.

Dit ADT kan worden gebruikt in een aantal verschillende contexten. Hiervoor wordt het ADT uitgebreid met meer functies. Waarbij die extra functies specifiek zijn voor de context waarin de ADT gebruikt gaat worden. Dit wordt in de nu volgende paragrafen besproken. Eerst wordt in § 2.3.1 nog even formeel vastgelegd wat een State Transformer is.



Figuur 10 : Een samenstelling.

2.3.1 Wat is een State Transformer

Een State Transformer is een berekening die een toestand, van het type *s*, transformeert en een resultaat van het type *a*[13] oplevert. Deze toestand is zodanig dat de referential transparency van functionele talen volledig behouden blijft. Zoals uit de inleiding blijkt, zijn State Transformers op zichzelf niet nieuw en waren ze altijd al mogelijk in functionele talen. Het heeft er al die tijd ingezeten, maar is niet eerder gebruikt.

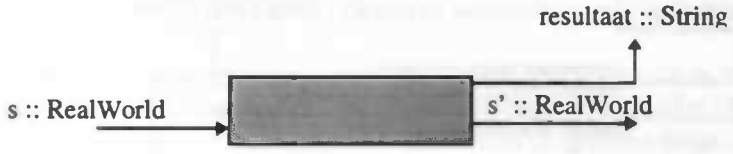
Het type van een State Transformer is gedefinieerd als: `ST s a`. Dit is een data-constructor met twee argumenten. Een voorbeeld hiervan is de volgende typesynoniem definitie :

```
type IO a = ST RealWorld a.
```
[14]

Dit is een State Transformer die een state van het type `RealWorld` gebruikt. Een ander voorbeeld, gebruik makend van de definitie van IO, is :

```
type IO_String = IO String = ST RealWorld String .
```

Een grafische voorstelling is :



Figuur 11 : Een grafische weergave van de State Transformer `IO_String`.

Met State Transformers is het mogelijk om toestand in het programma aan te brengen. De gevolgen hiervan zijn :

- Variabelen, zoals in een imperatieve taal, zijn mogelijk. § 2.3.2.
- Expliciete sequentiering in een programma is mogelijk. § 2.3.3.
- I/O is mogelijk. § 2.3.4.
- Compilers kunnen efficiënter worden. § 2.3.5.

2.3.2 Variabelen met State Transformers

Met een State Transformer kunnen variabelen gebruikt worden in een functioneel programma. Dit gebruik is zonder verlies van de belangrijke eigenschap van functionele talen, namelijk referential transparency.

Standaard aanwezig moeten zijn de functies returnST en thenST. Voor het werken met variabelen zijn een aantal standaard State Transformers gedefinieerd[15] . De naamgeving is overgenomen van [7].

---

[13] *s* en *a* zijn type variabelen

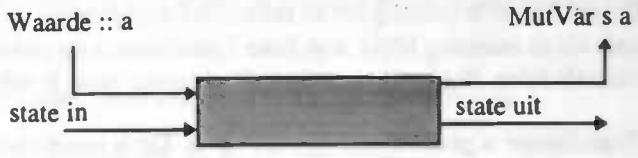
[14] Deze definitie van IO wordt gebruikt in Gofer

[15] Deze State Transformers worden gebruikt in Gofer. Zie § 3.4 en § 9.1 voor de verschillen met Gofer en de GHC-compiler.

- `newVar :: a -> ST s (MutVar s a )`

  Deze State Transformer voegt aan een state van het type *s* een referentie naar een waarde van het type *a* toe. Het resultaat van deze State Transformer is :
  - een aangepaste state van het type s waarin de waarde van het type a is opgeslagen
  - de pointer naar deze waarde. Voor de pointer wordt gebruik gemaakt van de data-constructor `MutVar s a`. Deze constructor wordt later besproken.

  Een grafische voorstelling van deze State Transformer staat in figuur 12.



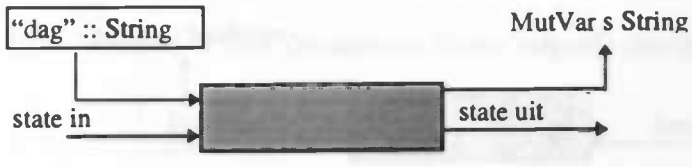
Figuur 12 : Grafische voorstelling van `newVar`.

Een voorbeeld voor het gebruik van `newVar` is :

```
newVar "dag"
```

Het type van die voorbeeld is : `String -> ST s (MutVar s String)`

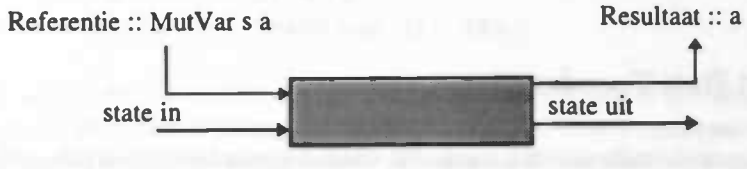
De grafische voorstelling van dit voorbeeld is :



Figuur 13 : Grafische voorstelling van `newVar "dag"`.

- `readVar :: MutVar s a -> ST s a`

Deze State Transformer kan gebruikt worden om de waarde van een referentie op te leveren. `readVar x` zal de waarde van *x* opleveren in een State Transformer. De state blijft onveranderd. Een grafische voorstelling van deze State Transformer is :

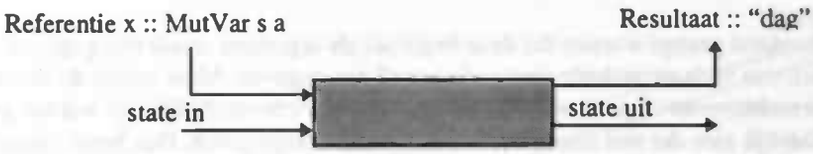


Figuur 14 : Grafische voorstelling van `readVar`.

Een voorbeeld is :

```
newVar "dag" `thenST` \x -> readVar x
```

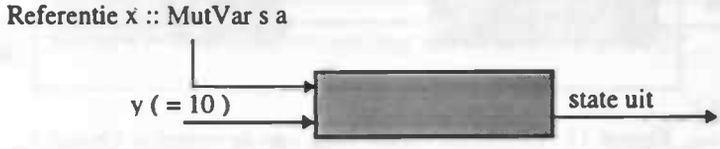
Grafisch is dit voorbeeld te zien als in figuur 15.



Figuur 15 : Grafische voorstelling van het voorbeeld.

- `writeVar :: MutVar s a -> a -> ST s ()`

Met de State Transformer `writeVar x y` zal de state worden veranderd zodanig dat de referentie `x` op de waarde `y` wordt afgebeeld. Er wordt geen resultaat opgeleverd[16]. Bijvoorbeeld `writeVar x 10` betekend dat `x` in de state `s` de waarde 10 zal hebben. Uit de typering van deze expressie blijkt dat `x` van het type `MutVar s Int` moet zijn, `y` van het type `Int` en `s` een type state. Grafisch is het gebruik van `writeVar` in het voorbeeld te zien in figuur 16.



Figuur 16 : Grafische voorstelling van het voorbeeld.

Met deze vier State Transformers kan gewerkt worden met variabelen. Deze State Transformers kunnen worden uitgebreid naar bijvoorbeeld arrays. Een voorbeeld daarvan is te vinden in [7,hoofdstuk 3].

Belangrijkste conclusie is dat variabelen in een functioneel programma mogelijk is, zonder de voordelen van functionele talen aan te tasten. Het is dus een echte positieve toevoeging aan functionele talen.

2.3.3 Sequentiering met State Transformers

Met State Transformers in een functionele taal is het mogelijk om een volgorde aan te brengen in het evalueren van een expressie. In een gewone functionele taal is dit niet mogelijk. Dit is een gevolg van het declaratieve karakter van functionele talen. Er wordt in een functioneel programma aangegeven wat gedaan moet worden en niet in welke volgorde.

In een imperatief programma wordt de volgorde impliciet aangegeven door de ";". Iets dergelijks is nu ook in functionele talen mogelijk door gebruik van State Transformers. Dit wordt bereikt door State Transformers te combineren. En zoals bekend is dat de functie `thenST`. In deze paragraaf een aantal andere sequentierings functies.

[16] Als geen resultaat opgeleverd wordt het type van het resultaat genoteerd met (), bijv ST s ().
Een State Transformer ST s () wordt gebruikt om alleen de state aan te passen.

Een vraag die opkomt is het volgende : is `thenST` nu de sequentie van een imperatieve taal zoals die bijvoorbeeld in Pascal in de vorm van de ";" voorkomt, of toch niet. De semantiek van de opeenvolging in Pascal is informeel als volgt :

stat_1 ";" stat_2

⇔

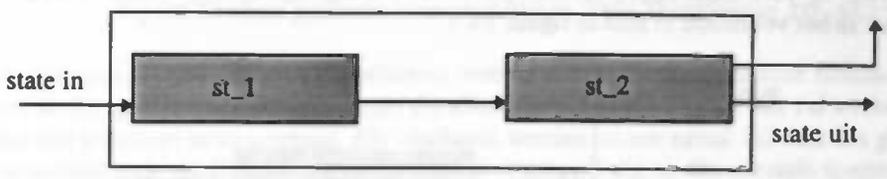
voer s1 uit voor s2 waarbij zowel s1 als s2 de toestand kunnen veranderen.

Er kan over de toestand gezegd worden dat deze impliciet als argument wordt meegegeven. Namelijk,als s1 de toestand verandert, dan is de veranderde toestand voor s2 een gegeven. Maar tussen de beide statements wordt geen resultaat rechtstreeks uitgewisseld. Uit de typering van `thenST` blijkt dit wel het geval te zijn. En ook figuur 9 laat duidelijk zien dat niet alleen de toestand wordt doorgegeven. Dus komt `thenST` niet overeen met de ";" uit Pascal.

De volgende State Transformer komt wel overeen met de ";" uit Pascal.

```
thenST_ :: ST s () -> ST s b -> ST s b
st_1 `thenST` st_2 = st_1 `thenST` \_ -> st_2
```

Een grafische voorstelling staat in figuur 17.



Figuur 17 : Grafische voorstelling van de operator `thenST_`.

Andere voorbeelden van combinerende State Transformers die een volgorde afdwingen zijn :

- ```
  listST  :: [ST s a ] -> ST s [a]
  listSt sts = foldr consST nilST sts
               where nilST       = return []
                     consST m ms = m  `thenST` \r ->
                                   ms `thenST` \rs -> return (r:rs)
  ```

  De functie `listST` neemt een lijst van State Transformers en maakt er één State Transformer van door de elementen van de lijst samen te voegen en de resultaten van iedere State Transformer samen te voegen in een lijst (figuur 18).

- ```
  listST_ :: [ST s a]  -> ST s ()
  listST_ sts = foldr (>>) (return ()) sts
  ```

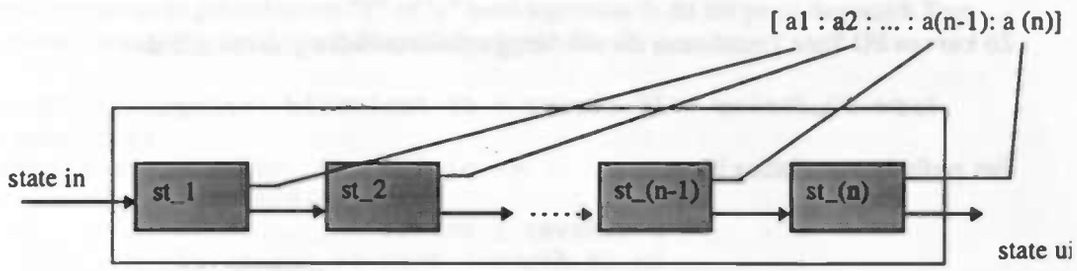
  De functie `listST_` neemt een lijst van State Transformers en maakt er één State Transformer van door de elementen van de lijst samen te voegen. In iedere afzonderlijke State Transformer wordt geen resultaat opgeleverd. Deze State Transformer past alleen de state aan. Een grafische weergave is figuur 19.

- ```
  mapST   :: (a -> ST s b)  -> [a] -> ST s [b]
  ```

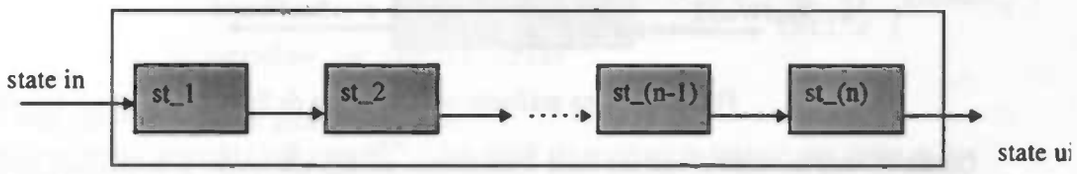
  Deze functie is in grote mate gelijk aan `listST`. Op het resultaat van één State Transformer wordt alleen nog een functie toegepast. In figuur 20 is dit uitgebeeld.
  ```

- `mapST_ :: (a -> ST s ()) -> [a] -> ST s ()`

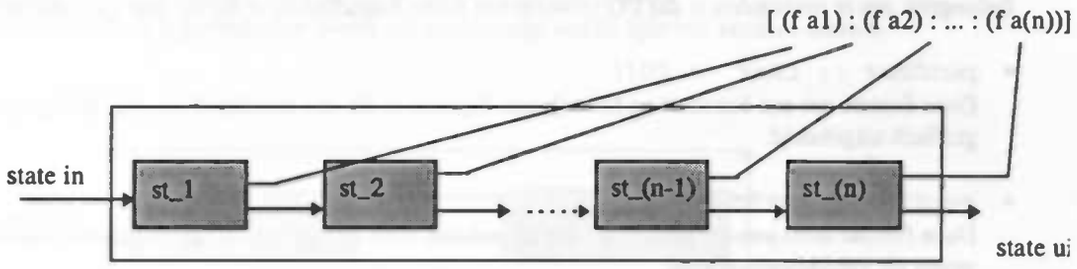
  Deze functie past op een lijst van State Transformers op ieder element een functie toe en levert geen resultaat op. Alleen de state wordt aangepast. Dit is in figuur 21 uitgebeeld.



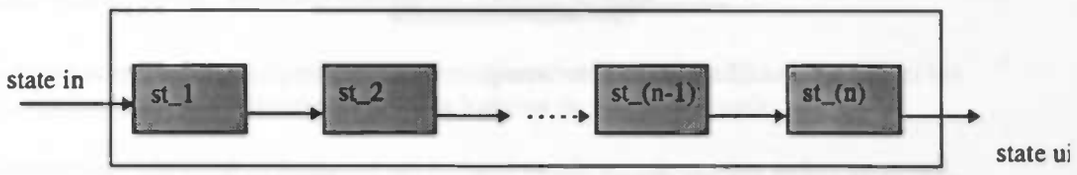
Figuur 18 : Grafische voorstelling van de operator `listST`.



Figuur 19 : Grafische voorstelling van de operator `listST_`.



Figuur 20 : Grafische voorstelling van de operator *mapST*.



Figuur 21 : Grafische voorstelling van de operator *mapST_*.

2.3.4 I/O met State Transformers

Met het creëren van een toestand in een State Transformer is het mogelijk om I/O te plegen. Omdat de toestand in een State Transformer geen side-effects kent is het gedeelte van het programma waarin I/O plaats vindt ook nog steeds volledig functioneel. Anders gezegd, er kan I/O worden gepleegd op een volledig functionele manier. In tegenstelling tot sommige huidige talen waarin het I/O deel van het programma niet meer volledig functioneel is.

I/O is een State Transformer zoals alle andere, alleen de toestand waarmee gewerkt wordt is van een speciaal type. In Gofer is deze toestand een abstract type genaamd `Realworld`. In § 2.3.1 is dit type al genoemd. Dit abstracte type is een modelering van de "buiten wereld".

Voor I/O wordt is het volgende type-synoniem gedefinieerd :

```
type IO a = ST RealWorld a.
```
[17]

Zo kan een I/O State Transformer die een String oplevert worden gedefinieerd als :

```
type IO_String = IO String = ST RealWorld String .
```

Een grafische voorstelling is :

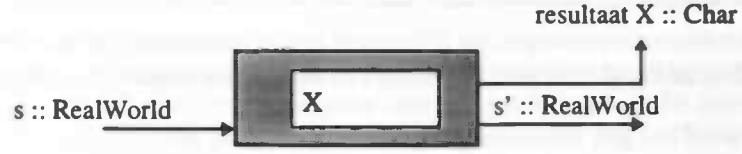
Figuur 22 : Een grafische weergave van de State Transformer `IO_String`.

Omdat I/O een instantie is van een State Transformer werken alle eerder gedefinieerde functies ook op I/O. Voor I/O zijn een aantal specifieke functies gemaakt die alleen op de toestand `RealWorld` werken. Deze operaties zijn dus niet polymorf in de toestand. Als voorbeeld worden nu een aantal I/O functies getoond. Belangrijk om te onthouden is dat I/O gewoon een State Transformer is die op één speciale toestand werkt.

- `putChar :: Char -> IO()`
  Deze functie zet een karakter op het scherm. Meestal is dit een standaard functie. In figuur 23 wordt dit grafisch uitgebeeld.

- `getChar :: IO(Char)`
  Deze functie leest een karakter van het toetsenbord. Ook dit is meestal een standaard functie. In figuur 24 wordt dit grafisch uitgebeeld.

Figuur 23 : Een grafische weergave van de functie `putChar`. In s' van het type `RealWorld` is s zo aangepast dat X op het scherm staat.



Figuur 24 : Een grafische weergave van de functie `getChar`. In s' van het type `RealWorld` is s zo aangepast dat X van het toetsenbord ingelezen is.

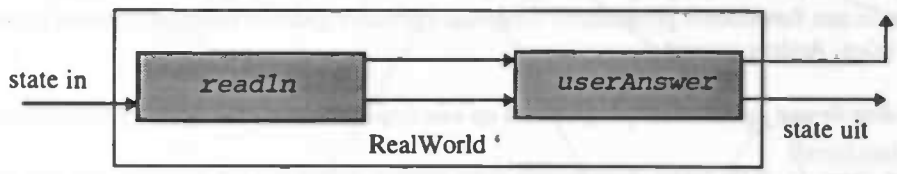
[17] Deze definitie van IO wordt gebruikt in Gofer

- ```
  userAnswer :: IO Bool
  userAnswer = readln `thenST` \answer ->
               if (toUpperString answer) == "Y"
               then return (True)
               else return (False)
  ```

  Deze functie bepaalt of de gebruiker een "Y" of "y" heeft ingetoetst. Is dit het geval dan wordt True opgeleverd, anders False. In figuur 25 wordt dit grafisch uitgebeeld.

- ```
  readln :: IO String
  readln = readit []
    where readit x = getChar `thenST` \c ->
                     case ord(c) of
                     10        -> return ( reverse x )
                     8         -> case (length x) of
                                    0            -> readit x
                                    otherwise    -> putChar ' ' `thenST_`
                                                    putChar (chr 8) `thenST_`
                                                    readit ( tail x )
                     otherwise -> readit (c:x)
  ```

  Deze functie leest een regel in die de gebruiker intoetst.
  - Wordt de Enter toets ingedrukt (ord c == 10^{18}) dan wordt als resultaat de inverse van de tot nu toe ingelezen lijst toetsen opgeleverd.
  - Wordt de Back Space ingedrukt (ord c == 8) dan wordt een teken gewist. Als er tenminste nog tekens te wissen zijn. Dit wordt bepaald door de lengte van de lijst met toetsen.
  - Wordt een andere toets ingedrukt dan wordt die toegevoegd aan de lijst van eerdere toetsen.



Figuur 25 : Grafische voorstelling van `userAnswer` en `readln`.

2.3.5 Efficiëntie van de programma's met State Transformers

Een punt dat even kort genoemd gaat worden is dat State Transformers de mogelijkheid bieden tot het efficiënter maken van functionele programma's. Vergelijk hiervoor de volgende situatie :

> Stel dat een expressie wordt geëvalueerd aan de hand van de waarde van een andere expressie. Bijvoorbeeld het vervangen van de eerste waarde in een lijst door een andere waarde. In een functionele taal moet dan eerst een copy worden gemaakt van de bronexpressie en die copy kan worden gebruikt voor het evalueren van de doel expressie. Dit is nodig omdat de eerste waarde in zijn bereik overal hetzelfde moet zijn en blijven. Zou de eerste waarde worden aangepast, dan is inherent sprake van een toestand, want in de loop van de tijd verandert de waarde. Dit impliceert het hebben van een toestand en side-effects.
>
> Stel nu dat de bronexpressie groot is, in executie tijd en/of geheugen ruimte gemeten. Dan vraagt het maken van een copy ook veel tijd en ruimte. Met een State Transformer is state mogelijk zonder side-effects en kan de waarde in-place aangepast worden. Er hoeft geen copy gemaakt te worden. Dit levert aanzienlijke mogelijkheden op om de efficiëntie van functionele programma's te vergroten.

[18] Deze nummers gelden voor DOS. Voor LINUX gelden andere getallen.

Een voorbeeld hiervan staat in [4] met arrays. In dit voorbeeld blijkt echter dat het nog niet zover is. Maar de potentie tot efficiëntere programma's is wel aanwezig.

Een andere bron voor verhoging van de efficiëntie ligt in het sequentieren van de evaluatie. Omdat de compiler weet dat bij, bijvoorbeeld de `thenST` operator, eerst de eerste berekening en dan de tweede berekening moet worden uitgevoerd, kan daarvan gebruik gemaakt worden. Op een andere manier bekeken :

Een van de voordelen van functionele programma's is het declaratieve karakter ervan. De compiler moet maar uitzoeken in welke volgorde de expressie geëvalueerd moet worden. De programmeur hoeft zich daar niet mee bezig houden. Maar, diezelfde programmeur heeft er ook geen enkele controle over. De volgorde van evalueren die de compiler kiest hoeft niet de snelste te zijn.

Imperatieve talen zijn juist het tegenovergestelde. Hier bepaalt de programmeur exact de volgorde van uitvoering van de statements. En kan zo proberen een snellere volgorde te vinden.

Meestal zal de programmeur de evaluatie volgorde aan de compiler over willen laten. Dit is juist een van de voordelen van functionele programma's. Maar soms niet, omdat hij / zij controle wil hebben over die evaluatie om bijvoorbeeld de snelheid te verhogen. Dit kan nu met State Transformers.

2.3.6 Conclusie State Transformers

State Transformers maken toestand mogelijk en daarmee variabelen, I/O en sequentiering. Het gebruik van State Transformers in de huidige vorm is, van uit syntactisch oogpunt bekeken, soms nog redelijk ingewikkeld. Als State Transformers doordringen in de functioneel wereld is speciale syntax eventueel te overwegen. In de commandoshell zijn ook enige functie gemaakt waarmee het gebruik eenvoudiger wordt.

2.4 FUNCTIONELE EN IMPERATIEVE PROGRAMMEER TALEN

Een korte beschouwing van de verschillen tussen functionele en imperatieve talen. Met State Transformers kan toestand in een functioneel programma gebracht worden. Hierdoor zijn variabelen, I/O en sequentiering van evaluatie in een functioneel programma mogelijk. Hierdoor gaan de functionele talen richting de imperatieve talen. Anders gezegd :

"er kan in een functioneel programma op een imperatieve manier geprogrammeerd worden".

Of dit een zegen is, is de vraag. Functionele talen hebben voordelen ten opzichte van dan imperatieve. Dus als op imperatieve wijze geprogrammeerd gaat worden in een functionele taal dan blijven mogelijkheden onbenut liggen.

Imperatieve talen hebben echter ook voordelen. Dus is het imperatief programmeren in een functionele taal soms ook aantrekkelijk.

De praktijk zal leren in welke richting de ontwikkelingen zullen gaan. Denkbaar is dat er een synthese van beide talen komt. Met de voordelen van functionele talen en imperatieve talen.

2.5 FUNCTIONELE EN LOGISCHE PROGRAMMEERTALEN

Om alle systemen te kunnen beschrijven is eigenlijk een logische programmeertaal nodig. Deze zijn echter nog niet snel genoeg om te gebruiken. Mede daarom is gekozen voor een functionele taal. Maar het probleem van snelheid gold ooit ook eens voor functionele talen. En het is denkbaar dat logische talen in de toekomst snel genoeg zullen zijn. Is in dat geval de gemaakte programmatuur nog wel bruikbaar. Met andere woorden, wordt niet een te grote beperking gelegd door de keuze van een functionele programmeertaal. Waarschijnlijk niet omdat functionele talen een onderdeel kunnen zijn van logische talen. Een voorbeeld is LIFE (Logic, Inheritance, Functions and Equations).

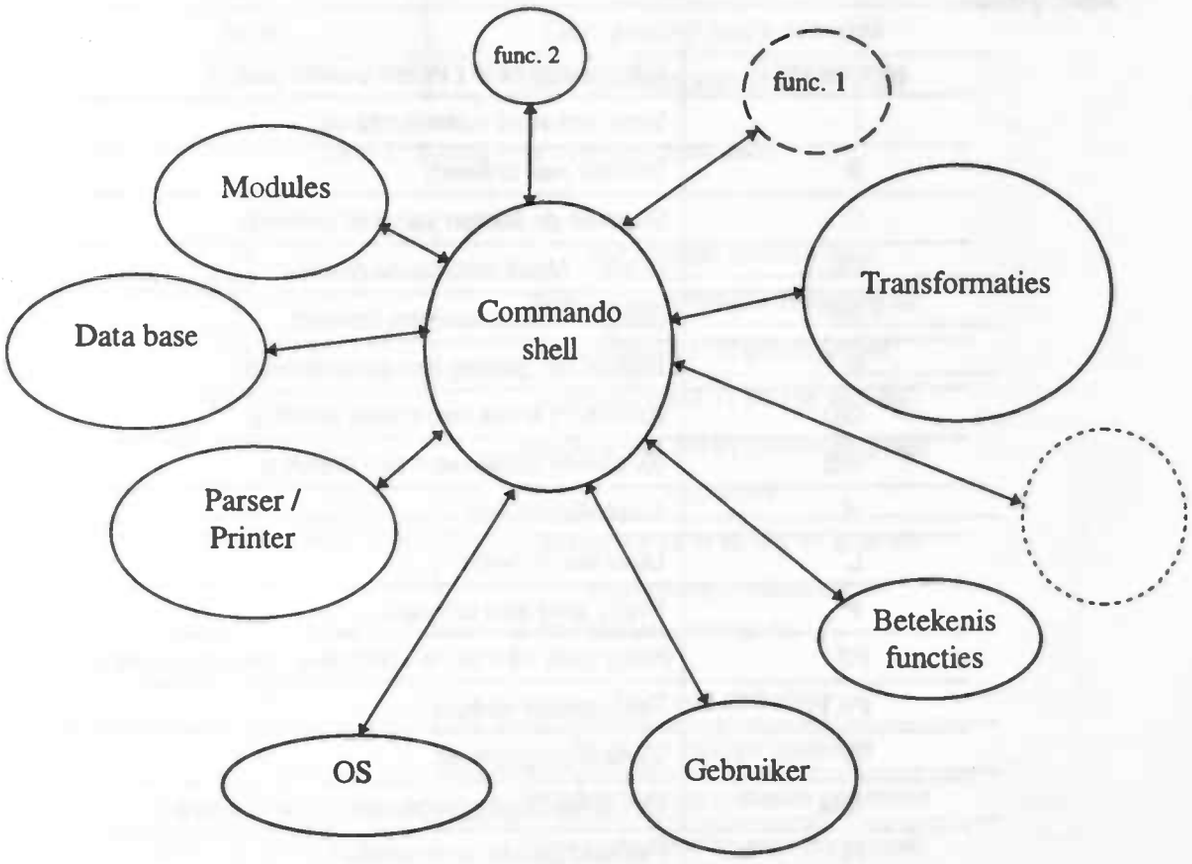
Vanuit theoretisch oogpunt hebben logische talen de voorkeur maar vanuit praktisch oogpunt hebben functionele talen de voorkeur. Gelukkig is deze voorkeur niet echt beperkend.

3 DE COMMANDOSHELL

3.1 INLEIDING

De ontwikkelomgeving wordt gemaakt om de ideeën van het onderzoek concreet te maken. Een belangrijk onderdeel uit deze ontwikkelomgeving is de commandoshell. Deze shell verzorgt de interactie met de gebruiker, het operating systeem en de onderdelen van de ontwikkelomgeving. In figuur 26 is dit uitgebeeld.

De commandoshell is te zien als een Integrated Development Enivorment (IDE) voor de ontwikkelomgeving. De applicaties, bijvoorbeeld een transformatie proces uit de Transformational Design, "draaien" op de commandoshell.



Figuur 26 : Een schematische weergave van de plaats van de commandoshell.

De volgende eisen zijn gesteld aan de implementatie van commandoshell :

- aan de commandoshell moeten gemakkelijk nieuwe functies toegevoegd kunnen worden. Deze eis is gesteld omdat meerdere mensen onderdelen gaan maken voor de ontwikkelomgeving. Om te voorkomen dat deze mensen eerst de commandoshell moeten leren kennen voordat ze een nieuw onderdeel kunnen toevoegen moet het toevoegen eenvoudig zijn. Om dit te bereiken is de opzet van de commandoshell sterk modulair met een paar centrale modules waar de overige modules gebruik van maken.
- de commandoshell moet geschreven worden in een functionele taal. De redenen hiervoor zijn :
  - De voordelen van het programmeren in functionele talen.
  - Onderzoeken van de nieuwe ontwikkelingen in functionele talen. Met name op het gebied van het weg werken van de nadelen van functionele talen.

- De verbinding met de overige begrippen uit de ontwikkelomgeving waarin functies een grote rol spelen.

In dit hoofdstuk wordt een inleiding gegeven in de commandoshell. Begonnen wordt met de commando's die de shell herkent. Hiermee wordt vanuit de gebruikerskant naar de shell gekeken zodat ook direct duidelijk is waar over gesproken wordt. In de volgende paragraaf wordt de commandoshell opgedeeld in onderdelen (de verschillende cirkels in figuur 26). Van deze onderdelen wordt aangegeven welke modules gebruikt worden in het onderdeel.

3.2 DE COMMANDO SET

In tabel 1 staan de commando's die de commandoshell herkent. Veel van deze commando's kunnen nog verder geparameteriseerd worden. Een aantal voorbeelden staan in de tabellen 2 tot 7. Voor een compleet overzicht wordt verwezen naar de helpfile. Tijdens het werken met de shell kan de syntax van een commando worden opgevraagd door "-?" achter het commando in te voeren. De invoer mag zowel in kleine als grote letters gebeuren.

commando -?	Uitgebreide help (indien beschikbaar)
!	Voer een shell commando uit
B	Bewaar een ontwerp
C	Bereken de kosten van een ontwerp
FD	{File} Maak een nieuw ontwerp
FE	{File} Editen van een ontwerp
G	Bepaal het gedrag van een ontwerp
GD	{Grafisch} Maak een nieuw ontwerp
GE	{Grafisch} Editen van een ontwerp
H	Geef History lijst
L	Laad een ontwerp
P	Pretty print een ontwerp
PA	Pretty print de interne weergave van een ontwerp
PAT	Testpatroon maken
Q	Verlaat programma
QM	Quine McClusky toepassen op een ontwerp
R	Herhaal laatste commando
RM	Verwijder (onderdeel uit een) file
S	Set omgevings parameters
SIM	Voer een simulatie uit
SS	Geef lijst met ingelezen systemen
ST	Statistics van programma
X	Maak Xilinx uitvoer van een ontwerp

Tabel 1 : Commando set.

c -A name cost	Voeg een nieuwe default kosten toe
c -L	Geef de lijstmet default kosten
c -R name	Verdwijder een element
c name_1.. name-n	Bereken de kosten van name_1 ....

Tabel 2 : Syntax voor Kosten commando.

ss	Toon complete state
ss -F	Toon geladen files
ss -S	Toon geladen systemen
ss -T	Toon geladen tuples
ss -B	Toon geladen basis definities
ss -A	Toon geladen atomaire definities

Tabel 3 : Syntax voor overzicht commando.

s	Toon huidige environment
s -s	Bewaar environment in setting.lst
s v1	Verwijder v1 uit environment
s v1 v2	Voeg het paar v1 v2 toe aan de environment als v1 daarin nog niet voorkomt
s v1 v2	Veranderd v1 in v2 als v1 al in de environment voorkomt

Tabel 4 : Syntax voor environment commando.

p -@	Print alle definities
p -C	Print het huidige systeem
p -C -O	Print het huidige systeem geordend
p { -F [-O] | -S | -T | -B | -A } name	Print onderdeel "name" (eventueel gesorteerd)
p -F { -S | -T | -B | -A } namef name	Print onderdeel "name" uit file "namef"

Tabel 5 : Syntax voor print commando.

r	Herhaal laatste commando
r !! number	Herhaal commando uit de history lijst

Tabel 6 : Syntax voor herhaal commando.

q	Beëindig programma
q s	Beëindig programma en bewaar veranderde systemen

Tabel 7 : Syntax voor einde commando.

3.3 OPBOUW COMMANDOSHELL

De commandoshell is grofweg in te delen in de volgende onderdelen :

- een parser voor het inlezen van systemen
- een printer voor het tonen van ingelezen systemen
- een database met functies op de database
- een help file
- een hoofdprogramma met een oneindige herhaling voor het inlezen en uitvoeren van gewenste commando's
- ondersteunende functies
- een verzameling commando's :
    - lees een systeem
    - verwijderen een systeem
    - bewaar een systeem
    - print een systeem
    - geef de history lijst
    - maak een testpatroon
    - maak Xilinx uitvoer
    - ....

De commandoshell is uitgebouwd uit een aantal modules. Aan de hand van de modules wordt deze onderverdeling :

- parser : **sslexer.hs parsefun.hs ssprbase.hs sspratom.hs ssparser.hs**
- printer : **ssiseq.hs sspp.hs ssadpp.hs**
- database : **commdata.hs commproj.hs**
- help file : **commhelp.hs**
- datastructuren : **commdata.hs commmain.hs ssad.hs patdata.hs xilidata.hs**
- Xilinx : **xilieval.hs xililam.hs xilinorm.hs xiligen.hs**
- testpatroon : **patstate.hs patpars.hs patfunc.hs patputil.hs patwutil.hs pattmain.hs**
- hoofdprogramma : **comm.hs commrdln.hs**
- utilities : **ssutil.hs commutil.hs compghc.hs ssad.hs**
- verzameling commando's :
    - systeem inlezen : **commload.hs**
    - verwijderen systeem : **commremv.hs**
    - bewaar systeem : **commsave.hs**
    - print systeem : **commprad.hs commpret.hs**
    - history lijst : **commhist.hs**
    - ....

In tabel 8 staat de hele lijst van modules. In deze tabel komt eerst de naam van de module. Dan wat de module "doet". En als laatste, als de module een implementatie van een commando is, wordt aangegeven bij welk commando uit de commando set de module hoort.

comm.hs	Module met het hoofdprogramma en commando "herhaal"	r
commcost.hs	Module voor het commando "bereken kosten"	c
commdata.hs	Module met database definities.	
commdesg.hs	Module voor het commando "{file} beschrijf systeem".	fd
commedit.hs	Module voor het commando "{file} verander systeem".	fe
commhelp.hs	Module met de help functies.	
commhgdb.hs	Module met database queries.	
commhist.hs	Module voor het commando "history".	h
commload.hs	Module voor het commando "laden van een systeem".	l
commlsys.hs	Module voor het commando "toon onderdelen van database".	ss
commmain.hs	Module met datatypen voor de commandoshell.	
commpatt.hs	Module voor het commando "test patroon"	pat
commprad.hs	Module voor het commando "pretty printen van onderdelen in de interne data structuur".	pa
commpret.hs	Module voor het commando "pretty printen van onderdelen".	p
commproj.hs	Module met de projectie- en selectiefuncties voor de database	
commquit.hs	Module voor het commando "einde programma".	q
commrdln.hs	Module met functies voor het lezen van gebruikers invoer vanaf het toetsenbord	
commremv.hs	Module voor het commando "verwijderen van systemen".	rm
commrest.hs	Module met rest commando's.	
commsave.hs	Module voor het commando "bewaren van systemen".	b
commset.hs	Module voor het commando "set".	s
commshel.hs	Module voor het commando "shell"	!
commtrns.hs	Module voor het commando "transformaties".	
commutil.hs	Module met een verzameling hulp functies voor de commandoshell.	
commxili.hs	Module voor het commando "xilinx uitvoer"	
compghc.hs	Module voor compatibiliteit tussen Gofer en GHC, de GHC file.	
compgof.hs	Module voor compatibiliteit tussen Gofer en GHC, de Gofer file.	
costmain.hs	Module voor het commando "bereken kosten"	
gdraw.hs	Module voor commando "grafische editor".	gd
parsefun.hs	Module met algemene parser functies.	
patdata.hs	Module met datatypen voor testpatroon programma	
patfunc.hs	Module voor het maken van uitvoer van testpatroon programma	
patpars.hs	Module met parser voor testpatroon files	
patputil.hs	Module met utilities voor het testpatroon programma	
patstate.hs	Module met datatypen voor testpatroon programma	

pattmain.hs	Module met hoofdprogramma voor testpatroon programma	
patwutil.hs	Module met window-utilities voor het testpatroon programma	
ssad.hs	Data structuur van de Systeem Subset.	
ssadpp.hs	Module met een pretty printer van een systeem waarbij de uitvoer een geformatteerde weergave is van de interne data structuur.	
ssiseq.hs	Module met een ADT waarmee string uitvoer geformatteerd kan worden.	
sslexer.hs	Module met scanner functies.	
ssparser.hs	Module met hoofd parser functie.	
sspp.hs	Module met een pretty printer van een systeem waarbij de uitvoer een geformatteerde weergave is in de grammatica van de Systeem Subset.	
sspratom.hs	Module met een parser voor een atomaire definitie.	
ssprbase.hs	Module met een parser voor een basis type-defintie.	
ssutil.hs	Module met een verzameling hulpfuncties.	
win.c	Code voor implementatie van X - functies.	
Win.hs	Module met Haskell functies voor aanroepen X - functies.	
win.hs	Module met dummy functies voor aantroepen X - functies voor Gofer	
xilidata.hs	Module met datatypen voor testpatroon programma	
xilieval.hs	Module met expressie evaluator voor "xilinx uitvoer"	
xiligen.hs	Module met generatie-functies voor "xilinx uitvoer"	
xililam.hs	Module met lambda-lifter voor "xilinx uitvoer"	
xilinorm.hs	Module met normalisator voor "xilinx uitvoer"	

Tabel 8 : Overzicht van de gebruikte modules.

3.4 HASKELL : GOFER EN DE GLASGOW HASKELL COMPILER (GHC-COMPILER)[19]

De commandoshell is geschreven in de functionele taal Haskell. Voor het ontwikkelen van de commandoshell is gebruik gemaakt van de interpreter Gofer 2.30 B met de files **cc.prelude**, **iomonad.gs**, en **ioarray.gs**. Dit is de versie met State Transformers.

Om de executie snelheid te vergroten is het programma gecompileerd met de Glasgow Haskell Compiler, de GHC-compiler, versie 0.25. Bij het compileren is, daar waar nodig, gebruik gemaakt van de voorlopige uitgave van Haskell 1.3. Wanneer de Haskell 1.3 commissie definitief besluit hoe Haskell 1.3 er uit komt te zien kan blijken dat sommige stukken code veranderd moeten worden. Er is echter zoveel mogelijk gebruik gemaakt van de delen die waarschijnlijk niet gaan veranderen.

Tussen Gofer en de GHC-compiler zijn de nodige verschillen[20] . Deze verschillen hebben hoofdzakelijk te maken met State Transformers en dan nog specifiek met de syntax. Hetzelfde kan worden uitgedrukt, alleen op een andere manier. Verder, en dat is wel fundamenteel anders, kent de GHC-compiler twee typen I/O :

- een eenvoudige, *primIO* genaamd. Dit is de *IO* van Gofer.
- een luxe, *IO* genaamd. Dit is een uitbreiding *primIO* en wordt gebruikt voor fout afhandeling.

[19] Vanaf deze paragraaf wordt het verslag concreet en zal de implementatie van de commandoshell besproken worden.

[20] Voor de verschillen tussen Gofer en de GHC-compiler, zie ook § 9.1.

Doordat de GHC-compiler deze twee typen IO kent, zijn er ook twee versies van de standaard State Transformer functies. Zo zijn er `returnPrimIO` en `return`, `thenPrimIO` en `>>=`.

Het voert te ver om alles op dit punt te behandelen. Maar opmerkelijk is het dat iedereen hetzelfde bedoelt maar alleen anders formuleert. Met de komst van de definitieve versie van Haskell 1.3 en het uitkristalliseren van onder andere de State Transformers zal uiteindelijk wel een stabiele situatie ontstaan. De uiteindelijke informatie voor de commandoshell is gevonden in :

- [3]
- [6]
- [7]
- [bibliotheken van GHC-compiler]

In de volgende tabel worden een aantal verschillen tussen Gofer en de GHC-compiler genoemd :

Gofer	GHC
thenST	>>=
thenST_	>>
IO	primIO
_	IO
_	primIOToIO
_	ioToPrimIO
MutVar	MutableVar
openfile	readFile
returnST	return
putString	putStr
putchar	putChar
getchar	getChar

Tabel 9 : Enkele verschillen tussen Gofer en de GHC-compiler.

Het is dus al met al redelijk in beweging in de Haskell wereld. Het enige advies dat gegeven kan worden aan toekomstige programmeurs is

- kijk naar al bestaande code (bijvoorbeeld deze commandoshell)
- zoek naar overeenkomende namen en kijk naar de typering van de verschillende functies. Dit helpt vaak om te zien of twee functies gelijk *kunnen* zijn.

3.5 TOEVOEGEN VAN EEN NIEUW ONDERDEEL

Omdat veel nog ontwikkeld moet worden is het van belang dat een nieuw onderdeel eenvoudig toe te voegen is. Door de modulaire opbouw is dit, mijns inziens, bereikt. De handelingen voor het toevoegen van een nieuw onderdeel zijn :

1. een nieuwe tak in de data-constructor Command uit **`commmain.hs`** toevoegen.
2. een nieuwe tak toevoegen aan de case uit selectcommand uit **`commrdln.hs`**.
3. een nieuwe help functie toe voegen in **`commhelp.hs`** voor het nieuwe onderdeel.
4. uitbreiden van de algemene help functie, *`helpcom`* uit **`commhelp.hs`**, zodat bij het opvragen van de mogelijke commando's het nieuwe onderdeel ook getoond wordt.
5. een nieuwe tak in de functie *`select_action`* uit **`comm.hs`**. Zowel voor de help-tak als de uitvoer-tak.
6. importeren van de nieuwe module(s) in **`comm.hs`**.

7. de makefile aanpassen.
8. make aanroepen en wachten (soms nog een beetje langer wachten)

4 DE SYSTEEM SUBSET

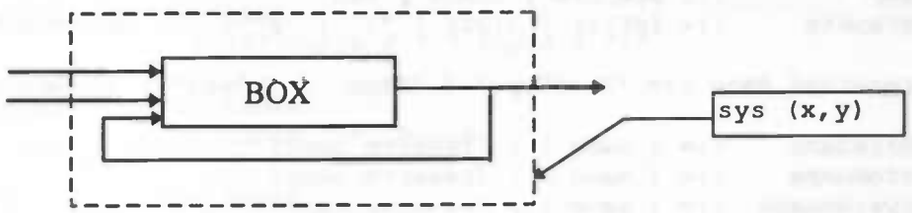
De beschrijving van systemen gebeurt met functies. Er is een grammatica gemaakt waaraan deze beschrijvingen moeten voldoen. Deze grammatica wordt de Systeem Subset genoemd en is het onderwerp van dit hoofdstuk. De Systeem Subset is een onderdeel van de implementatie van de System Semantics uit § 1.4.

4.1 INLEIDING

Een systeem wordt beschreven als een zin uit de Systeem Subset. Deze Systeem Subset is een grammatica waarin de volgende concepten mogelijk zijn :

- Functie applicatie.
- Het benoemen van een element uit het systeem.

In het voorbeeld uit figuur 27 is de functie applicatie het "verbinden" van de verschillende componenten. Met het benoemen van een systeem element is het mogelijk terugkoppelingen aan te brengen. Of onderdelen van het systeem een keer te definiëren en vaker te gebruiken in andere systemen.




```
sys (x,y) = (box(x,y,sys(x,y))
```

Figuur 27 : Een voorbeeld uit de Systeem Subset.

Deze twee concepten zijn genoeg voor het beschrijven van de in het onderzoek beschouwde systemen. Maar voor het handig werken met de Systeem Subset is ook nog enige syntactic sugar toegevoegd, zoals tuples in tuples.

Voor de naam *"Systeem Subset"* is gekozen omdat dat de Systeem Subset een beperking ("subset") is van een typische functionele taal en de beperking zodanig gekozen is dat hiermee systemen beschreven kunnen worden.

Van de functionele taal zijn een aantal mogelijkheden verwijderd zoals bijvoorbeeld pattern-matching. Een systeem dat beschreven is in de Systeem Subset kan, module syntactische conversies, ook beschreven worden in een typische functionele taal. Andersom kan niet altijd. De beperking van een functionele taal tot de Systeem Subset kan ook als volgt gedefinieerd worden :

> ***"De Systeem Subset is dat deel van een functionele taal dat geïnterpreteerd kan worden als een verzameling van verbonden fysieke componenten ".***

4.2 DE SYNTAX VAN DE SYSTEEM SUBSET

De grammatica staat in de figuren 27 en 28 in een BNF-syntax. In figuur 28 wordt de lexicale opbouw van de tokens beschreven.

De volgende notatie is gebruikt :

- Een string c tussen `".."` betekent dat deze string letterlijk voor moet komen in de invoer
- Een element tussen `{..}` komt 0 of meer keren voor.
- Een element tussen `[ .. ]` komt 0 of 1 keer voor.
- De `|` geeft een keuze weer.
- De `/` is de verzamelings verschil operatie.

4.2.1 Lexicale opbouw van de tokens

De lexicale opbouw van de tokens staat in figuur 28.

```
small       ::= "a" | "b" | ... | "z"
large       ::= "A" | "B" | ... | "Z"
letter      ::= small | large | "_"
digit       ::= "0" | "1" | ... | "9"

name        ::= letter { letter | digit }

whitespace  ::= whitestuff { whitestuff }
whitestuff  ::= whitechar | comment | longcomment
whitechar   ::= newline | space | tab | vertab | formfeed

comment     ::= "--" { any } newline
     longcomment ::= "{-" { any } "-}"
any         ::= graphic | space | tab
graphic     ::= letter | digit | "!" | "$" | ... enzovoort

reserved_name ::= "BaseType" | "Atom" | "where" | "endwhere"

basename    ::= ( name ) / {reserve_name}
atomname    ::= ( name ) / {reserve_name}
systemname  ::= ( name ) / {reserve_name}
```

Figuur 28 : Lexical opbouw van de tokens.

4.2.2 De grammatica in BNF-notatie.

Bij het begin van deze afstudeeropdracht werd gewerkt met de grammatica uit figuur 29. Deze grammatica laat duidelijk de structuur zien. Maar het blijkt dat deze grammatica niet goed te parsen is omdat die niet *LL(1)* is. In figuur 30 is een equivalente grammatica te zien die wel *LL(1)* is. Met deze grammatica gaat gewerkt worden. In de onderstaande bespreking wordt nog gewerkt met de grammatica uit figuur 29 omdat die duidelijker is.

```
text          ::= elem { ";" elem }

elem          ::= base_type_decls
                | atom_defs
                | sys_defs

base_type_decls ::= "Basetype" basename { "," basename }

atom_defs     ::= "Atom" a_def { "," a_def }
a_def         ::= atomname "::" systype

sys_defs      ::= sys_def { ";" sys_def  }
sys_def       ::= [ systemname "::" systype ";" ]
                    systemname [arg] "=" exp [ "where" sys_defs

                "endwhere" ]
                | [ systemname "::" systype ";" ]
                    tuple              "=" exp [ "where" sys_defs
                                                   "endwhere" ]

tuple         ::= name
                | "(" tuple { "," tuple } ")"

systype       ::= tuple
                | tuple "->" tuple

arg           ::= tuple

exp           ::= aexp { aexp }
                | "\" tuple "->" exp

aexp =        ::= name
                | "(" exp { "," exp } ")"
```

Figuur 29 : De oorspronkelijke grammatica van de systeem subset.

Een beschrijving van een systeem bestaat op toplevel niveau uit elementen met daartussen een `";"` als scheiding[21]. Een element kan een van de vier volgende dingen zijn :

- Een basis type-defintie waarmee typen gedefinieerd worden. Deze typen worden later gebruikt om het soort verbinding tussen de componenten weer te geven. Dit kan bijvoorbeeld zijn :
  - Boolean verbinding
  - Integers verbinding
  - Optische verbinding
  - .....

  of zelf een pneumatische verbinding in het geval van een (deels) pneumatisch systeem.

  Een voorbeeld van een basis type-defintie is : `"BaseType 01,02"`.

  De opbouw van een basis type-defintie is : eerst het keyword `"BaseType"`, dan een of meer identifiers gescheiden door een `","`.

[21] Dit is een verschil met een functionele taal omdat daarin de scheiding tussen de verschillende elementen geregeld wordt met de lay-out rule.

- een atomaire definitie waarmee atomaire functies gedefinieerd worden. Deze functies kunnen gezien worden als "built in" functies.

  Een voorbeeld van een atomaire definitie is : `"Atom foo :: B -> B".`

  De opbouw van een atomaire definitie is : eerst het keyword `"Atom"`, dan een identifier, dan de string `"::"` en dan een typering van de functie zoals in functionele talen gebruikelijk is.

  Bij het type van de atomaire functie blijkt ook direct de beperking van de Systeem Subset ten opzichte van een functionele taal. Het type kan alleen van de vorm : `(domein) -> (bereik)` zijn.

- Een systeem definitie.

  Een voorbeeld van een complete systeem definitie is :

  ```
  f1 :: B -> B; f1 a = not a where ... endwhere.
  ```

  Een systeem definitie begint met de naam van de functie, een identifier. Dan volgt de string `"::"` en een typering van domein en bereik. Dan volgt een ";" waarna de echte definitie van het systeem komt. Eerst weer de naam, dan de argumenten, dan het "=" symbool en dan een expressie met eventueel een where-clausule.

  Dit is een complete systeem definitie. Het eerste deel van de typering is optioneel. Het voorbeeld mag ook zijn :

  ```
  f1 a = not a.
  ```

- een tuple definitie.

  Een tuple definitie lijkt sterk op een systeem definitie. In het onderstaande voorbeeld, f2,

  ```
  f2 :: B; f2 = not where ... endwhere.
  ```

  zijn de verschillen ten opzichte van f1 :
  - er zijn geen argumenten
  - f2 in het tweede deel van de definitie is een tuple
  - de typering kent alleen een domein.

Een tuple in de Systeem Subset is de gebruikelijke tuple. Bijvoorbeeld `a`, `(a,b)` , `(a, (b,c))` etc. De argumenten van een systeem definitie en de typering van domein en bereik zijn ook van deze vorm.

Een expressie is :
- een rij van atomaire expressies, bijvoorbeeld : `and True False`
- een lambda expressie, bijvoorbeeld : `\x -> maal 2 x`

De opbouw hiervan is duidelijk.

Een atomaire expressie is een van de volgenden :
- een naam, bijvoorbeeld : `and`
- een of meer expressies gescheiden door "," tussen haken, bijvoorbeeld : `( True, False )`

4.2.3 De equivalente grammatica

In figuur 30 staat de equivalente grammatica die *LL(1)* is. Een grammatica is *LL(1)* wanneer :

voor elke twee paar producties met gelijke linkerkant A geldt :

$$La(A \rightarrow a) \cap La(A \rightarrow b) = \emptyset$$

Hierin is $La(A \rightarrow b)$ de verzameling lookahead symbolen van de produktie $(A \rightarrow b)$.

Deze vezameling $La(A \rightarrow b)$ is gedefinieerd als :

$La(A \rightarrow b) =$ *if nullable* $(A \rightarrow b)$
then
$first(A \rightarrow b) \cup follow(A \rightarrow b)$
else
$first(A \rightarrow b)$

met de functies *nullable*, *first* en *follow* gedefinieerd als op pagina 103 tot 109, § 4.5 uit [12]

Met deze functies is bepaald wat de La's zijn van de producties. Dit staat in de grammatica genoteerd met :

```
==> LA = { ... }
```

De notatie {- ... -} en {+ ... +} is sterk verwant aan de lookahead-sets. Met {-... -} wordt aangegeven voor welke tokens een herhaling in de grammatica moet ophouden. Met {+ ... +} wordt aangegeven wanneer de herhaling voortgezet moet worden.

```
start             ::= text "Bottom"

text              ::= elem { ";" elem }   ==> {+ ";" +}
                                          ==> {- "Bottom" -}

elem              ::= base_type_decls     ==> LA = {"BaseType"}
                    | atom_defs           ==> LA = {"Atom"}
                    | s_dt

base_type_decls   ::= "Basetype" name { "," name }
                                          ==> {+ "," +}
                                          ==> {- ";" , "Bottom" -}

atom_defs         ::= "Atom" atom_def { "," atom_def }
                                          ==> {+ "," +}
                                          ==> {- ";" , "Bottom" -}

atom_def          ::= name "::" systype

s_ds              ::= s_dt { ";" s_ds }   ==> {+ ";" +}
                                          ==> {- "endwhere" -}
```

Figuur 30 : De equivalente grammatica waarmee gewerkt gaat worden.

```
st_d              ::= name st_d1
                    | tuple "=" exp td_2  ==> LA = { "(" }

st_d1             ::= "::" tuple st_d2    ==> LA = { "::" }
                    | "=" exp td_2        ==> LA = { "=" }
                    | arg "=" exp sd

st_d2             ::= "->" tuple ";" name arg "=" exp sd
                                          ==> LA = { "->" }
                    | ";" tuple "=" exp td_2
                                          ==> LA = { ";" }

sd                ::= "where" s_ds "endwhere"
                                          ==> LA = { "where" }
                    | Empty
                                          ==> LA = Follow ( sd_2 )
                                                 = { ";" , "Bottom" }

tuple             ::= name                ==> LA = { ident }
                    | "(" tuple { "," tuple }")"
                                          ==> {+ "," +}
                                          ==> {- ")" -}
                                          ==> LA = { "(" }

arg               ::= tuple

exp               ::=  aexp { aexp }      ==> {+ ident , "(" +}
                                          ==> {- "where", ",", ")",
                                                 "Bottom" , ";"      -}
                    | "\" tuple "->" exp  ==> LA = { "\" }

aexp              ::=  name
                    | "(" exp { "," exp } ")"
                                          ==> {+ "," +}
                                          ==> {- ")" -}
                                          ==> LA = { "(" }
```

Figuur 30 vervolg : De equivalente grammatica waarmee gewerkt gaat worden.

4.3 VOORBEELDEN VAN ZINNEN UIT DE SYSTEEM SUBSET

Een paar eenvoudige zinnen uit de taal van de Systeem Subset laat zien wat voor zinnen er gemaakt kunnen worden.

```
BaseType B;
BaseType A,E;

Atom and :: ( B,B ) -> ( B, B ), or :: ( B,B ) -> B;
Atom  f1 :: B -> B;
Atom  f2 :: B -> ( B,B );

bolt (  ( i0,i1,i2,i3 ),
        ( i4,i5,i6,i7 ),
        ( i8,i9,i10,i11 ),
        ( i12,i13,i14,i15) ) =
       nor4 ( a,b,c,d ) where a = or4 ( i0,i1,i2,i3)
                                where i0 = true;
```

```
                                        i1 = false;
                                        i2 = true;
                                        i3 = false
                                    endwhere;
                                b = or4 ( i4,i5,i6,i7);
                                c = or4 ( i8,i9,i10,i11);
                                d = or4 ( i12,i13,i14,i15)
                            endwhere;

halfadder (x,y) = (exor (x,y), and(x,y));

fulladder (x,y,c) = (som,carry)
                    where carry   = or (a,b);
                          (d,b)   = halfadder (x,y);
                          (som,a) = halfadder (d,c);
                    endhwere;

ser_add (a,b) = sum
                where (sum,c) = full_adder (a,b,y);
                          y = dff c
                endwhere;

abs4 :: INT -> INT;
abs4 a = min een
                where een = min vijf vier
                      where vier = maal twee twee;
                            vijf = deel dertig zes
                      endwhere
                endwhere
```

5 DE DATASTRUCTUUR VOOR DE SYSTEEM SUBSET

Een schakeling wordt beschreven in de taal van de Systeem subset. Om er mee te kunnen werken in de ontwikkelomgeving moet een schakeling ingelezen en verwerkt worden in een interne datastructuur. Deze datastructuur is het onderwerp van dit hoofdstuk. Eerst wordt de structuur getoond en besproken. Daarna worden enkele voorbeelden van omzettingen van een zin uit de Systeem subset in de data structuur uitgewerkt. De data structuur staat in de file **ssad.hs**.

5.1 DE DATASTRUCTUUR

De datastructuur wordt gebruikt voor een interne weergave van een systeem. Variabelen van dit type worden gebruikt voor het opslaan van een ingelezen systeem. De datastructuur staat in figuur 31.

```
type Tekst a    =  [Elem a]

data Elem  a    =  Base a
                |  Atom a
                |  Sys_def (a, NTuple a, Expr a, Tekst a)
                |  Tuple_def (a, NTuple a, Expr a, Tekst a)

data NTuple a   =  Ident a
                |  NTuple [NTuple a]

data Expr a     =  EName a
                |  EAp (Expr  a)  (Expr a)
                |  ELam (NTuple a) (Expr a)
                |  ATuple [Expr a]

data Name       = Id IdDetails IdType

data IdDetails =  TupleWithNoName
                |  TupleWithName String
                |  BaseName      String
                |  AtomName      String
                |  FunctionName  String
                |  ArgName       String
                |  TupleName     String

type GenDomain a = (NTuple a)
type GenRange  a = (NTuple a)

data IdType     =  NoType
                |  AtomDef SSDomain SSRange
                |  SysDef SSDomain SSRange
                |  TupleDef SSDomain
```

Figuur 31: De datastructuur voor de interne opslag.

Deze datastructuur is een directe vertaling van de grammatica uit figuur 29. Een grondige bespreking blijft daarom achterwege. Twee opmerkingen.

Een systeemdefinitie wordt opgeslagen als een 4-tuple. Het eerste element is de naam, het tweede element zijn de argumenten, het derde element is de expressie en het vierde element zijn de eventuele where-clausules.

Zijn er geen where-clausules, dan is dit een lege lijst. Een tuple definitie lijkt hier sterk op. Het tweede element echter is nu het tuple en het eerste element is de naam, als die gegeven is. Zie ook het volgende.

De data-constructor `IdDetails` kent voor de verschillende soorten strings een aparte tak. Met name voor de tuples zijn er een aantal verschillende vormen. De tak `TupleWithNoName` wordt gebruikt bij een tupledefinitie waaraan geen naam toe te kennen is. De tak `TupleWithName` wordt gebruikt bij een tupledefinitie waaraan wel een naam toe te kennen is. De tak `TupleName` wordt gebruikt om een string op te slaan die een onderdeel is van een tuple. Voorbeelden van de verschillen zijn :

- `(a,b,c) = .......`

Laat dit een definitie zijn. Het is dan een tuple definitie zonder expliciete typering. Het is niet duidelijk hoe hier eenduidig een naam aan toe te kennen is. Hier wordt daarom gekozen voor de volgende interne representatie :

```
[Tuple_def (Id TupleWithNoName).....]
```

- `a :: (B,B) ; (e,d) = .....`

Als dit een definitie is, dan is dit weer een tuple-definitie. Maar nu is het type expliciet opgegeven. Dan moet ook een naam worden gegeven in het eerste deel. Deze naam is dan te gebruiken als naam van de definitie. Let wel, de naam *a* staat nu voor het tuple `(e,d)`. Er staat eigenlijk :

```
a := (e,d).
```

- `de (a,b,c) = .....`

Nu is het tuple (a,b,c) een "echt" tuple. Dit wordt dan opgeslagen als :

```
.....(Ident (TupleName a) _ ) ....
```

Met de functie *PA* uit de commandoshell kan een nette weergave van de interne toestand gegeven worden zodat een en ander bestudeerd kan worden.

5.2 ACCESS-FUNCTIES VOOR DE DATASTRUCTUUR

Bij de datastructuur zijn een aantal access-functies gemaakt die de verschillende onderdelen uit de datastructuur halen. Deze functies zijn erg gemakkelijk. Wanneer men bekend is met de naamgeving van de functies, dan weet met ook gelijk wat ze doen[22].

Om in de typering van functies te kunnen zien wat "de semantiek" van het argument is, zijn type-synoniemen gedefinieerd. Met semantiek wordt bedoeld dat aan het type te zien is wat het argument voorstelt. Bijvoorbeeld kan het type van een functie zijn :

```
f3 :: BaseName -> AtomName -> ArgName
```

Dan is het eerste argument de naam van een basis type-defintie, het tweede argument is de naam van een atomaire definitie en het resultaat is de naam van een argument. Zie bijvoorbeeld de file **`comproj.hs`** of de onderstaande access-functies voor gebruik van deze typen. Deze types worden alleen gebruikt als een geheugensteuntje, een conventie. Er wordt geen enkele controle uitgevoerd of er daadwerkelijk met bijvoorbeeld een naam van een basis type wordt aangeroepen[23].

Naast deze access-functies zijn een aantal "algemenere" functies gedefinieerd. Zo sorteert `sortall` een lijst van elementen in de volgorde :

- basis type-defintie.
- atomaire definities.

---

[22] ADS staat voor : Abstracte Data Structuur

[23] Dit zou kunnen wanneer voor de verschillende argumenten data constructoren gemaakt worden

getADSBaseName	Elem Name -> BaseName
getADSAtomName	Elem Name -> AtomName
getADSSysName	Elem Name -> SysName
getADSTupleName	Elem Name -> TupleName
getADSPartName	Elem Name -> String
getADSExpr	Elem Name -> Expr Name
getADSSysExpr	Elem Name -> Expr Name
getADSTupleExpr	Elem Name -> Expr Name
getADSWExpr	Elem Name -> [Elem Name]
getADSSysWExpr	Elem Name -> [Elem Name]
getADSTupleWExpr	Elem Name -> [Elem Name]
getADSArgs	Elem Name -> NTuple Name
getADSSysArgs	Elem Name -> NTuple Name
getADSTupleArgs	Elem Name -> NTuple Name
getADSType	Name -> IdType
getIdentName	NTuple Name -> String
getEName	Expr Name -> String
getName	Name -> String
getADSRange	Elem Name -> NTuple Name
makeArgList	Elem Name -> [String]
getRes	Elem Name -> [String]
tuple_elem	String -> NTupleName -> Bool
tuple_elem_list	String -> [NTuple Name] -> Bool
convert	NTuple Name -> [String]
deconvert	[String] -> NTuple Name
sortall	Tekst Name -> Tekst Name
isNoType	IdType -> Bool
isBaseDef	Elem Name -> Bool
isAtomDef	Elem Name -> Bool
isSysDef	Elem Name -> Bool
isTupleDef	Elem Name -> Bool

- systeem definities.
- tuple definities.

Hiervoor wordt gebruik gemaakt van de functies `isBaseDef`, `isAtomDef`, `isSysDef` en `isTupleDef`. Deze functies bepalen met patternmatching op de data-constructor `Elem a` welke tak van `Elem a` gebruikt is.

Twee andere functies zijn `convert` en `deconvert`. `Convert` haalt de strings uit een `NTuple a` en `deconvert` pakt een lijst met string in, in `NTuple a`.

De functies *getRes* en *makeArgList* maken gebruik van de functie *convert*. Met *getRes* wordt van een definitie het bereik bepaalt. Het resultaat van *getRes* is de lijst van string die samen het bereik vormen. Een voorbeeld :

```
getRes (B,I,N) = [ "B","I","N"]
```

Deze functie wordt gebruikt in het programma waarmee een testpatroon gemaakt kan worden, hoofdstuk 14. De functie *makeArgList* doet hetzelfde, alleen nu voor de argumenten van een definitie.

5.3 VOORBEELDEN VAN OPSLAG IN DE INTERNE DATASTRUCTUUR:

In dit gedeelte worden handmatig enige zinnen uit de Systeem Subset "geparst" en in de datastructuur § 5.1 gezet[24].

```
BaseType A,B,C,INT                                  <<==>>

[ Base (Id ( BaseName "A" ) NoType),
  Base (Id ( BaseName "B" ) NoType),
  Base (Id ( BaseName "C" ) NoType),
  Base (Id ( BaseName "INT" ) NoType)
]

Atom  and :: ( B, B) -> B                           <<==>>

[ Atom (Id ( AtomName "and" )
           ( AtomDef ( NTuple [  Ident ( Id ( BaseName "B" ) NoType ),
                                 Ident ( Id ( BaseName "B" ) NoType )
                              ]
                     )
                     ( Ident ( Id ( BaseName "B" ) NoType  ))
           )
      ),
  Atom (Id ( AtomName "or" )
           ( AtomDef ( NTuple [  Ident ( Id ( BaseName "B" ) NoType ),
                                 Ident ( Id ( BaseName "B" ) NoType )
                              ]
                     )
                     ( Ident ( Id ( BaseName "B" ) NoType  ))
           )
      )
]

abs a = een                                         <<==>>

[ Sys_defs [ ( Id ( FunctionName "abs" ) NoType,
               Ident ( Id ( ArgName "a" ) NoType),
               EName ( Id ( FunctionName "een" ) NoType ),
               []
             )
           ]
]
```

[24] Om van een groter systeem een weergave te krijgen kan in de commandoshell de optie *PA* gekozen worden.

```
abs :: INT -> INT; abs a = een                    <<==>>

[ Sys_defs [ ( Id ( FunctionName "abs" )( SysDef
                  ( Ident ( Id ( BaseName "INT" ) NoType ) )
                  ( Ident ( Id ( BaseName "INT" ) NoType ) ) ),
               Ident ( Id ( ArgName "a" ) NoType),
               EName ( Id ( FunctionName "een" ) NoType ),
               []
             )
           ]
]
```

6 EEN PARSER EN EEN PRINTER.

Een van de mogelijkheden die de commandoshell biedt is het laden van een systeem beschrijving. Voor het laden van een systeem wordt de beschrijving geparst en verwerkt in de interne data-structuur. Verder is het mogelijk om een interne weergave weer terug te zetten in "tekst"-vorm. Daarvoor zijn twee functies gemaakt :

- Een parser functie om invoer vanuit een file te parsen en in de interne data-structuur te zetten.
- Een pretty-printing functie om een interne data-structuur netjes te printen naar het scherm of naar file.

6.1 DE PARSER VOOR DE SYSTEEM SUBSET.

De parser-functie beslaat een aantal modules. Deze zijn :

- **sslexer.hs** : Deze module bevat de lexer. Deze leest een string in en groepeert deze string in tokens.
- **parsefun.hs** : Deze module bevat algemene parser functies. Deze functies zijn zo gekozen dat ze overeenkomen met de constructies uit de BNF-notatie.
- **ssprbase.hs** : Deze module bevat parser functies voor de basis type-definties.
- **sspratom.hs** : Deze module bevat parser functies voor de atomaire definities
- **ssparser.hs** : Deze module bevat de hoofdfunctie voor het parsen.

6.1.1 De scanner.

De module **sslexer.hs** bevat drie lexers :

- *startlex* : Deze lexer wordt gebruikt voor het scannen van "programma"-tekst
- *startArgLex* : Deze lexer wordt gebruikt voor het scannen van gebruikers invoer
- *startSetlex.* : Deze lexer wordt gebruikt voor het scannen van de settings-file "setting.lst"

De functie *startlex* levert een rij *ExtendedTokens* af. Een *ExtendedToken* is een tuple waarin behalve de string-representatie van het token ook het regel- en kolom-nummer van het token staat. Dit voor een goede foutmelding als een syntax fout in de invoer gevonden wordt. De rest van de functies spreekt voor zich.

De functie *startSetlex* is speciaal. Omdat de gebruiker voor de environment ook de volgende alias wil kunnen invoeren :

```
set padb "pa -db test",
```

waarbij de string tussen de "..." als één geheel moet worden beschouwd, moet hiervoor een speciale voorziening zijn. Want de "gewone" lexer, bijvoorbeeld startArglex, maakt hier van :

```
["set","padb","pa","-db","test"]
```

Als de gebruiker deze alias op de volgende manier invoert, maakt startSetlex er een goede opdeling van.

```
set {padb pa -db test}
```

Hiervoor maakt startSetlex gebruik van de functie dropUntil en words_ext.

type Token	String
type ExtendedToken	(String,Int,Int)
dropUntil	(a -> Bool) -> [a] -> [a]
words_ext	String -> [String]
startlex	String -> [ExtendedToken]
startArglex	String -> [Token]
startSetlex	String -> [Token]
lc	String -> Int -> Int -> [ExtendedToken]
ownlex	String -> Int -> Int -> [ExtendedToken]
isIdChar	Char -> Bool
isWhiteSpace	Char -> Bool
notWhiteSpace	Char -> Bool
twoCharOps	[String]

Tabel 10 : Functies uit de module **sslexer.hs**.

6.1.2 De algemene parser functies

In [13] en in de hbc-bibliotheek van de GHC-compiler worden algemene parser functies beschreven waarmee de voorkomende constructies van een BNF-notatie geparst kunnen worden. Met beide modules is geprobeerd een parser voor de Systeem Subset te maken.

Dit werkte voor de functies uit [13] niet goed. Bij een langere invoer duurde het al onwerkbaar lang voordat er een antwoord kwam. En dat terwijl de grammatica $LL(1)$[25] is en een ontleding dus in lineare tijd, ten opzichte van het aantal tokens, gedaan kan worden .

Tabel 11 laat zien hoe lang het duurde voordat de systeem beschrijving uit figuur 32 geparst was in Gofer[26] met de parser functies uit [13]. De invoer file bestond uit een aantal dezelfde definities achter elkaar. Er werd steeds een definitie toegevoegd aan de file waarna gekeken werd hoeveel reducties er nodig waren voor het parsen van de file.

Een opmerking bij het aantal reducties is dat het aantal reducties in de tabellen niet alleen gebruikt is voor het parsen. Een aantal reducties is gebruikt voor het opstarten van de commandoshell. Maar naarmate het aantal reducties groter wordt, is dit aantal relatief te verwaarlozen.

```
abs5 ( a ) = een
 where een = min vijf vier
                   where vier = maal twee twee;
                         vijf = deel dertig zes
                   endwhere
endwhere
```

Figuur 32 : Test definitie voor tabel 11.

[25] De gebruikte grammatica is de grammatica uit figuur 30.

[26] Gofer geeft aan het eind van het evalueren het aantal stappen, het aantal gebruikte cellen en het aantal garbage collections voor het evalueren van de expressies.

Aantal definties	Aantal reducties	#red n / #red 1	#red n / #red (n-1)	#red n - #red (n-1)
1	15247			
2	41764	2,739161802	2,74	26517
3	419552	27,51701974	10,05	377788
4	9207954	603,919066	21,95	8788402
5	21271669	1395,137994	2,31	12063715

Tabel 11 : Vergelijking van evaluatie kosten voor het systeem uit figuur 32.

Voor het systeem uit figuur 33 staan de resultaten in tabel 12.

```
abs5 ( a ) = een
 where een = min vijf vier
 endwhere
```

Figuur 33 : Test definitie voor tabel 11.

Aantal definities	Aantal reducties	#red n / #red 1	#red n / #red (n-1)	#red n - #red (n-1)
1	11194	1	0	
2	18053	1,61	1,61	6859
3	30086	2,69	1,67	12033
4	61848	5,53	2,06	31762
5	175020	15,64	2,83	113172
6	618474	55,25	3,53	443454
7	2379086	212,53	3,85	1760612
8	9409336	840,57	3,96	7030250
9	37519656	3351,76	3,99	28110320

Tabel 12 : Vergelijking van evaluatie kosten voor het systeem uit figuur 33.

Door het systeem nog eenvoudiger te maken, met het weg halen van de where-clausule, levert de parser goede resultaten.

Voor het systeem uit figuur 34 staan de resultaten in tabel 12.

```
abs5 ( a ) = min vijf vier
```

Figuur 34 : Test definitie voor tabel 12

Aantal definities	Aantal reducties	#red n / #red 1	#red n / #red (n-1)	#red n - #red (n-1)
1	5136	0	0	0
2	6265	1,22	1,22	1129
3	8075	1,57	1,29	1810
4	9333	1,82	1,16	1258
5	10867	2,12	1,16	1534
6	12401	2,41	1,14	1534
7	13935	2,71	1,12	1534
8	15469	3,01	1,11	1534
9	17003	3,31	1,10	1534

Tabel 13 : Vergelijking van evaluatie kosten voor het systeem uit figuur 34.

Uit de tabellen blijkt dat alleen systemen zonder where-clausule goed te parsen zijn. In de andere gevallen is het aantal reductie stappen exponentieel in het aantal definities. Dit heeft te maken met het non-determinisme in het algoritme van de parser. Bij een keuze in de grammatica (een `|` constructie in de BNF-notatie), wordt geen informatie gebruikt om te bepalen welke tak gevolgd moet worden. Om tot een antwoord te komen worden alle takken geparst, voorzover deze deeltakken goede ontledingen opleveren. Dit is als volgt weer te geven :

$$resultaat\ (p1 \mid p2 \mid .... \mid pn) = (resultaat\ p1)\ \%\ (resultaat\ p2)\ \%\ ..... \%\ (resultaat\ pn)$$

waarin % een operator is die een goede ontleding toevoegt aan de al eerder gevonden goede ontledingen. Omdat voor een beslissend antwoord alle ken geparst moeten worden, wordt veel onnodig werk verricht.

Dat mogelijk veel overbodig werk wordt verricht blijkt ook uit het type dat opgeleverd wordt. Een onderdeel in dit type is een lijst van mogelijke ontledingen en de daarbij behorende overgebleven tokens uit de invoer. Die ontledingen waarbij geen rest tokens over blijven worden goed genoemd. Het is dus mogelijk dat er meerdere ontledingen zijn. Maar de Systeem Subset moet niet ambigu zijn. Dit impliceert dat er dus maar een ontleding mogelijk is. Het opleveren van een lijst van mogelijke (deel-)ontledingen is dus niet nodig. Want de invoer is goed of fout.

Een laatste probleem bij de parser functies uit [13] was de slechte foutmelding. Of beter, de onmogelijkheid om een goede foutmelding te geven. De bepaling of een ontleding gelukt is of niet wordt gedaan aan de hand van de lijst van ontledingen en rest tokens. Als er geen ontleding gevonden werd zonder rest tokens dan is de ontleding mislukt.

Maar welke ontleding van de lijst geeft nu een goede indicatie van waar de fout is opgetreden. Dit is niet te bepalen. Er kan gekozen worden voor die ontleding met het minste aantal rest tokens. Maar dan nog blijft het gokken of dit de beste foutmelding geeft.

De parser functies uit de hbc bibliotheek werden gepresenteerd als snel, efficiënt en met goede foutmeldingen. Na bestudering van de code en na ettelijke pogingen om de functies te gebruiken is besloten dat ook deze functies niet bruikbaar zijn. De redenen hiervoor zijn :

- De opbouw en werkwijze van de functies zijn grotendeels gelijk aan die van de functies uit [13].
- Het type dat wordt opgeleverd is ook hier weer een lijst van ontledingen en resttokens.

Dus, ondanks dat de functies uit de hbc bibliotheek niet werkend gekregen zijn, wordt ook hier het slechtste verwacht ten aanzien van de snelheid van het ontleden. De foutmeldingen hadden wel de potentie om beter te zijn.

Besloten is toen om zelf algemene parser-functies te maken die werken met First en Follow informatie. Een parser gemaakt met deze parser-functies blijkt goed te werken. Deze algemene parser-functies staan in de module "**`parser.hs`**".

In de module wordt begonnen met het definiëren van een aantal typen en data constructoren. De meeste hiervan spreken voor zich. Belangrijk zijn het type `Parser` en de data-constructor `ParserResult`.

De constructor *ParserResult* bestaat uit twee alternatieven :

- *One a b.* : Deze tak wordt gebruikt als de ontleding geldig is. Het resultaat van een ontleding staat in het eerste element, de rest van de tokens staat in het tweede element. Omdat slechts een ontleding mogelijk is, is gekozen voor de naam One.
- *Fail String b* :
  Als er een fout optreedt wordt deze tak gebruikt. In het eerste element van het type String staat een foutmelding en in het tweede element staat de rest van de tokens.

Het type "*Parser a*" is een functie met de typering :

```
[ExtendedTokens] -> ParserResult a [ExtendedToken]
```

die nog geparameteriseerd kan worden met het type dat opgeleverd moet worden. Dit kan bijvoorbeeld

```
(Elem Name)
```

zijn. Dit wordt genoteerd met :

```
Parser (Elem Name).
```

Nu volgen de parser functies. Er zijn een aantal categorieën te onderscheiden :

- De functies *pLit* en *pVar* consumeren een token.
  - De functie *pLit* ontleedt een "Literal" token.
    Een produktieregel **A :: = ";"** wordt **p(A) = pLit ";".**
  - De functie *pVar* ontleedt een identifier.
    Een produktieregel **A ::= funcname** wordt **p(A) = pVar.**

  In deze beide functies wordt gecontroleerd of het einde van de file al bereikt is. Als dit zo is wordt een foutmelding gegeven.
- De functies *pAlt (x) (Simple)* worden gebruikt om een keuze in de grammatica te parsen. De x in pAlt(x) geeft aan hoeveel alternatieven er zijn. pAlt3 wil zeggen dat een keuze gemaakt moet worden uit drie alternatieven enz. In de pAlt(x) functies moet voor elk alternatief een LookAhead-Set gegeven worden. pAlt(x)Simple wil zeggen dat er een default regel is. Deze functies kunnen worden gebruikt wanneer niet voor alle alternatieven bepaalt kan worden wat de LookAhead -Set is. In deze functies wordt ook gecontroleerd of het einde van de file al bereikt is. Als dit zo is wordt een foutmelding gegeven.
- De functies *pThen (x)* staan voor een opeenvolging in een grammatica.
  Een produktieregel **A ::= x y z** wordt **p(A) = pThen3 {een functie} p(x) p(y)p(z).**
  De functie die direct na de pThen(x) komt is een functie die het resultaat van alle samenstellende parser samen voegt, een semantische functie. De (x) geeft het aantal elementen dat achter elkaar komt.
- *pOneOrMoreFixed, pZeroOrMoreFixed, pOneOrMoreWithSepFixed.*
  Deze functies zorgen voor de herhaling. De toevoeging "Fixed" geeft aan dat de beslissing om door te gaan met de herhaling of om te stoppen volledig wordt bepaald door twee LookAhead-Sets. In de grammatica uit § 4.2.3, figuur 30, wordt dit genoteerd met {+ ... +} en {- ... -}.

  Een voorbeeld :

  A ::= "(" B { "," B } ")" doorgaan ==> {+ "," +}
  stoppen ==> {- ")" -}

  wordt vertaald met

```
p(A) = pThen3 f ( pLit "(" )
                ( pOneOrMoreWithSepFixed p ( B ) [ "," ] ( pLit "," ) [ ")"] )
                ( pLit ")" )
```

  pOneOrMoreFixed is de Kleene-plus operatie. pZereOrMoreFixed is de Kleene-star operatie. pOneOrMoreWithSepFixed wordt gebruikt wanneer meerdere gelijke onderdelen, gescheiden door een seperator, ontleedt moeten worden. Zoals in het voorbeeld boven.
- *pOneOrMore, pOneOrMoreMinimal, pZeroOrMore, pOneOrMoreWithSep.*

Deze functies zijn vergelijkbaar met bovenstaande functies. Alleen is bij functies maar een LookAhead-Set nodig. In pOneOrMore, pZeroOrMore en pOneOrMoreWithSep is de LookAhead-Set voor het doorgaan met de herhaling nodig. In pOneOrMoreMinimal is de LookAhead-Set voor het stoppen van de herhaling nodig.

- `pApply`. Deze parser past op het resultaat van een parser een functie toe.
- `pEmpty`. Deze parser slaagt altijd zonder een token te consumeren. En levert als resultaat van de ontleding het argument van deze parser op.
- (>>>). Deze parser past een functie toe op het resultaat van een sequentie van twee parsers
- (+.+). Deze parser voert een sequentie van twee parsers uit. Het resultaat van beide parsers wordt opgeleverd als een 2-tuple (a,b) waarin a het resultaat van de eerste parser is en b het resultaat van de tweede parser.

6.1.3 De Parser.

In deze module wordt het grootste deel van de parser gedefinieerd. Hiervoor wordt gebruik gemaakt van de parser functies uit 6.1.2 en van de grammatica uit § 4.2.3, figuur 30.

De data-constructor `ParserValue` geeft aan of een ontleding gelukt is. Als deze gelukt is wordt de tak `Valid x` gekozen. De variabele `x` is de inhoud van de file in de interne weergave. Is de ontleding niet gelukt dan wordt de tak `NotValid x y` gekozen. De variabele `x` is het nummer van de regel waar de fout geconstateerd is. De variabele `y` is het kolomnummer in deze regel. De tak `Error` wordt gekozen als een andere fout optreedt, bijvoorbeeld wanneer een ingevoerde filename niet bestaat.

De data constructoren `InComplete(x)` worden gebruikt om tussentijds deel-informatie door te geven aan andere parser functies. De naamgeving is als volgt :

- `IC` betekent : In Compleet.
- `Ln` betekent : Een lange systeem of tuple definitie. Dit wil zeggen, een definitie met expliciete typering.
- `Sh` betekent : Een korte systeem of tuple definitie. Dus geen expliciete typering.

Deze aanpak lijkt ingewikkeld maar is nodig omdat iedere tak in een parser functie hetzelfde type op moet leveren. Dit kan worden bereikt met de data constructoren met verschillende alternatieven.

De functie `fp` is het startpunt van een ontleding. De invoer is een filenaam, de uitvoer is het resultaat van de ontleding, uitgedrukt in het type `ParserValue`.

Met `readAFile` wordt geprobeerd om de inhoud van een file te lezen. Het resultaat van `readAFile` is een 2-tuple (a,b) waarin a een boolean is die aangeeft of de file-operatie gelukt is. Is deze operatie gelukt dan is b de inhoud van de file. Is deze operatie niet gelukt dan is b een error-message.

Als er geen "file"-fout optreedt wordt de ontleding gestart door de inhoud van de file, geconcateneerd met "Bottom", aan de functie `ssparser` aan te bieden. Aan de hand van het resultaat wordt de tak gekozen van `ParserValue` en eventueel een error-message getoond.

De overige functies zijn directe vertalingen van de grammatica uit § 4.2.3, figuur 30.

- De functie `pssTekst` ontleedt een rij elementen gescheiden door een ;
- De functie `pssElem` ontleedt een element. Dus een van de drie mogelijkheden.
- De functies `pssST_d`, `pssST_d1` en functie `pssST_d2` worden gebruikt om delen van een systeem- of tuple-definitie te ontleden. In deze functies wordt door de data constructoren onderscheid gemaakt in de verschillende ontledingen. De stroom van informatie gaat vanuit de deelontledingen naar boven, naar pssST_d. Daar wordt een element van het type Elem Name opgeleverd.
- De functie `pssTuple` wordt gebruikt om een "echt" tuple te parsen.
- De functie `pssTuple_Type` wordt gebruikt om een tuple te parsen dat gebruikt is in een typering. Dit is eigenlijk dezelfde functie als `pssTuple`. Alleen nu wordt `BaseName` ingevuld bij `(Id (BaseName ..) ..)`.
- In de functie `pssArg` wordt een argument tuple geparst. Dit is eigenlijk dezelfde functie als `pssTuple`. Alleen nu wordt `ArgName` ingevuld bij `(Id (ArgName ..) ..)`.
- In `pssWhere` wordt een where-clausule geparst. Als deze where-clausule er niet is, wordt een lege lijst opgeleverd.

- In `pssExpr` wordt een expressie geparst. Dit kan zijn :
  - een rij Aexpr
  - een Lambda expressie
- In de functie `pssElam` wordt een lambda expressie geparst
- In de functie `pssAExpr` wordt een atomaire expressie geparst

6.1.4 De basis type-definties

In deze module is de functie voor het ontleden van een basis type-defintie. In `pssBase` worden een of meer basis type-defintie geparst en in de abstracte datastructuur geplaatst.
De hulpfuncties zijn :

- *mk_base* : deze maakt een nieuwe instantie van het datatype Base
- *mk_basedefs* : deze maakt van de nieuwe basis type-defintie een nieuwe instantie van `(Id (BaseName _) NoType)`

De rest is een rechtstreekse interpretatie van de grammatica uit § 4.2.3, figuur 30.

6.1.5 De atomaire definities

In deze module de functies voor het parsen van een atomaire definitie. De functies zijn :

- `pssAtom`. In `pssAtom` worden een of meer atomaire definities geparst en in de abstracte datastructuur geplaatst.
  De hulpfuncties zijn :
  - *mk_atoms* : deze maakt een nieuwe instantie aan van het datatype Atom
  - *make_atomdefs* : deze maakt van de nieuwe atomaire definities nieuwe instanties van `(Id (AtomName *) (AtomType ** ***))`
  - *make_atomtype* : deze maakt van een nieuwe atomaire definitie een nieuwe instantie van `(AtomType ** ***)`
- De functie `pssAtomDef` ontleedt één atomaire definitie. De naam wordt geparst, dan het teken :: en dan, met de funtie `pssAtomType`, het type van de atomaire definitie.
- De functie `pssAtomType` wordt gebruikt om het type van een atomaire definitie te ontleden. Eerst wordt het domein-tuple geparst, dan het bereik tuple. Er is van uitgegaan dat altijd een domein en een bereik gegeven moet worden. Dit is een goede eis omdat de bouwstenen waar wij mee werken van het type

  invoer → *uitvoer*

- zijn. De hulpfunctie *mk_atomtypedef* maakt een paar waarin het domein- en bereiktuple staan. In de functie `pssAtom` wordt dit paar weer ontleedt.
- De functie `pssTuple_AtomType` wordt gebruikt om een tuple te parsen. Het tuple hoort bij een atomaire type-defintie, en NIET bij een tuple definitie, zoals bijvoorbeeld (a,b) = (false, false). Dit onderscheid is gemaakt om in te kunnen vullen dat er een `BaseName` gebruikt is. Hulp functie is : `make_tuple` : Deze maakt een nieuwe instantie van het data-type `NTuple` en vult deze met een "echte" tuple.

6.2 DE PRETTY PRINTER VOOR DE SYSTEEM SUBSET.

De parser ontleedt een tekst, als die voldoet aan de Systeem Subset, en verwerkt deze tekst in een interne weergave. De pretty printer doet het tegenovergestelde. Een interne weergave van een systeem wordt omgezet in een tekstuele weergave. Er zijn twee pretty printers gemaakt. De eerste transformeert naar de gebruikelijke "programmatekst". Dit is de meest voor de hand liggende pretty printer. De tweede pretty printer transformeert de interne weergave naar een nette vorm van deze interne weergave. Deze pretty printer is gemaakt voor debug doeleinden. De interne toestand kan hiermee bekeken worden. De pretty printers zijn gebaseerd op de pretty printers uit [13]

6.2.1 Hulp functies en datastructuren voor de Pretty Printers

In [13] wordt een pretty printer beschreven voor de "Core language". Voor de efficiëntie wordt een datatype `Iseq` gemaakt. Dit datatype wordt ook in onze pretty printers gebruikt. De datatypen en functies staan in de file **`ssiseq.hs`**. Er zijn een aantal functies toegevoegd waarvan de werking voor zich spreekt. Zo is onder andere de functie `iDisplayIO` toegevoegd. Deze functie zet de string direct op het scherm door de string aan te bieden aan de functie `writeln`. Dit in tegenstelling tot de functie `iDisplay` dat een string oplevert. Voor de beschrijving van het datatype wordt verwezen naar [13].

6.2.2 De Pretty Printer voor "programmatekst"

De functies in deze module zijn :

- De functies `pp(1/2)(IO)` worden gebruikt om op top-level over te kunnen gaan op het datatype `Iseq` voor een tussentijdse representatie. pp(1/2) levert als resultaat een string op. pp(1/2)IO schijft deze string naar het scherm. Dit onderscheid is gemaakt voor het schrijven naar file, waarvoor pp(1/2) wordt gebruikt.
  - `pp1(IO)` verwerkt eerst de basis type-definties, dan de atomaire definities en dan de systeem- en tuple definities. Voor het sorteren wordt de functie `sortall` gebruikt uit de module **`ssutil.hs`**
  - `pp2(IO)` drukt de elementen in volgorde van voorkomen af
- De functie `ssprintGen` wordt gebruikt om op top-level niveau de functie `iDisplay` toe te kunnen passen. Tevens wordt hier het niveau van de expressies op 0 gezet. Dit niveau wordt gebruikt bij het afdrukken van expressies en where-clausules.
- In de functie `ssprTekst` wordt tussen de elementen een seperator geplaatst en de elementen zelf worden netjes geprint. De seperator is een ";" gevolgd door een newline. Op alle elementen wordt de functie `ssprElem` losgelaten door de map functie.
- In de functie ssprElem worden een element geprint. Dit is een van de vier mogelijkheden.
  - Een basis definitie
  - Een atomaire definitie
  - Een systeem definitie
  - Een tuple definitie
- In `ssprBase` wordt een Basis element geprint
- In `ssprAtom` wordt een atomaire definitie afgedrukt.
- In `ssprSysDef` wordt een systeem definitie afgedrukt.
- In `ssprTupleDef` wordt een tuple definitie afgedrukt.
- In de functie `ssprNTuple` wordt een tuple geprint. Er wordt onderscheid gemaakt naar alleen een naam of een tuple met haken
- In de functie `ssprName` wordt een identifier afgedrukt.
- In de functie `ssprTypeDef` wordt het type van een definitie afgedrukt.
- In de functie `ssprExpr` wordt een expressie geprint. Er wordt onderscheid gemaakt naar
  - Een naam
  - Een compositie
  - Een Lambda expressie
  - Een atomaire expressie
- In de functie `ssprWhere` wordt een where-clausule geprint.

De functies zijn opgebouwd naar de opbouw van de datastructuur uit § 5.1. Moeilijk zijn de functies niet. Wel lijken ze complex omdat met veel mogelijkheden rekening gehouden moet worden.

pp1	SSTekst -> String
pp2	SSTekst -> String
pp1IO	SSTekst -> IO()
pp2IO	SSTekst -> IO()
sspprintGen	(String -> Iseq) -> Tekst Name -> String
sspprTekst	(String -> Iseq) -> [Elem Name] -> Int -> Iseq
sspprElem	(String -> Iseq) -> Elem Name -> Int -> Iseq
sspprBase	(String -> Iseq) -> Name -> Iseq
sspprAtom	(String -> Iseq) -> Name -> Iseq
sspprSysDef	(String -> Iseq) -> (Name, NTuple Name, Expr Name, Tekst Name) ->Int -> Iseq
sspprTupleDef	(String -> Iseq) (Name, NTuple Name, Expr Name, [Elem Name]) ->Int ->Iseq
sspprNTuple	(String -> Iseq) -> NTuple Name -> Iseq
sspprName	(String -> Iseq) -> Name -> Iseq
sspprTypeDef	(String -> Iseq) -> IdType -> Iseq
sspprExpr	(String -> Iseq) -> Expr Name -> Int -> Iseq
sspprWhere	(String -> Iseq) -> Tekst Name -> Int -> Iseq

6.2.3 De Pretty Printer voor de datastructuur

Deze module is grotendeels gelijk aan de module uit § 6.2.2. De naamgeving van de functies is veranderd van ..*pp*.. in ..*adpp*.. . En de manier waarop de uitvoer gemaakt wordt is anders. Maar verder is alles gelijk. Daarom wordt deze module niet verder besproken.

adpp1	SSTekst -> String
adpp2	SSTekst -> String
adpp1IO	SSTekst -> IO ()
adpp2IO	SSTekst -> IO ()
ssadpprintGen	(String -> Iseq) -> [Elem Name] -> String
ssadpprTekst	(String -> Iseq) -> [Elem Name] -> Int -> Iseq
ssadpprElem	(String -> Iseq) -> Elem Name -> Int -> Iseq
ssadpprBase	(String -> Iseq) -> Name -> Iseq
ssadpprAtom	(String -> Iseq) -> Name -> Iseq
ssadpprSysDef	(String -> Iseq) (Name, NTuple Name, Expr Name, Tekst Name) -> Int ->Iseq
ssadpprTupleDef	(String -> Iseq) (Name, NTuple Name, Expr Name, Tekst Name) ->Int ->Iseq
ssadpprNTuple	(String -> Iseq) -> NTuple Name -> Iseq
ssadpprName	(String -> Iseq) -> Name -> Iseq
ssadpprTypeDef	(String -> Iseq) -> IdType -> Iseq
ssadpprExpr	(String -> Iseq) -> Expr Name -> Int -> Iseq
ssadpprWhere	(String -> Iseq) -> Tekst Name -> Int -> Iseq
printName	(String -> Iseq) -> String -> Iseq

7 DATABASE VAN DE COMMANDOSHELL

Met de database van de commandoshell wordt de ingelezen informatie beheerd. Onderdelen van de ontwikkelomgeving kunnen de database raadplegen via een verzameling "database"-functies. In § 7.1 worden de dataconstructoren en type synoniemen besproken van de database. In § 7.2 worden projectie functies van de type synoniemen besproken. In § 7.3 wordt algemene informatie gegeven over de database. In § 7.4 wordt voor ieder onderdeel van de database de database-functies behandeld.

De database bestaat uit de volgende modules :

- **commdata.hs**
- **commproj.hs**
- **commhgdb.hs**

7.1 DATA CONSTRUCTOREN EN TYPE SYNONIEMEN VOOR DE DATABASE

De database bestaat uit een data-constructor met 9 elementen waarin de informatie bewaard wordt. De elementen zijn :

(1) ⇒ **environment**.
Dit is de lokale environment.
(2) ⇒ **informatie**.
Dit is de verzameling ingelezen files.
(3) ⇒ **systeem definities**.
Dit is de verzameling ingelezen systeem definities.
(4) ⇒ **tuple definities**.
Dit is de verzameling ingelezen tuple definities.
(5) ⇒ **basis type-definities**.
Dit is de verzameling ingelezen basis type-definties.
(6) ⇒ **atomaire type-definities**.
Dit is de verzameling ingelezen atomaire definities.
(7) ⇒ **commando's**.
Dit is de verzameling ingevoerde commando's.
(8) ⇒ **standaard kosten definities**.
Dit is de verzameling standaard kosten van een aantal componenten.
(9) ⇒ **huidige systeem**.
Dit is de laatst ingelezen file.

```
data DesignData = DData [Env] [Info] [SysDef] [TupleDef] [BaseDef]
                       [AtomDef] [Commandd] [DefaultCost] CurSys
```

Uit deze definitie blijkt dat de database uit een aantal lijsten van een verschillende typen bestaat. Alleen *CurSys* is geen lijst. De elementen van de verschillende lijsten worden nu besproken.

1. Het type-synoniem *Env* wordt gebruikt om een, ten opzichte van het programma, lokale environment[27] op te slaan. Dit environment is een associatie lijst waarvan de elementen van het type *Env* zijn. *Env* is een 2-tuple waarvan beide elementen een string zijn. Het eerste element is de naam, de tweede is de waarde van naam. Via het "set-commando" kan deze environment geraadpleegd en veranderd kan worden.

[27] Met de GHC-compiler is het mogelijk om de environment van unix te benaderen. Het is hierdoor denkbaar dat uiteindelijk deze lokale environment niet nodig is. Dit zal de toekomst moeten uitwijzen.

```
type          Env = (EnvName,EnvValue)
type      EnvName = String
type     EnvValue = String
```

2. Een informatie item is een 2-tuple waarvan het eerste element een string is en het tweede element van het type *SSTekst* is. Het eerste element is een filename, het tweede element is de interne weergave van de inhoud van de file. In de file staan dus "systeem programma's" in de taal van de Systeem Subset.

```
type         Info = (FileName,InfoValue)
type    InfoValue = SSTekst
type     FileName = String
```

3. Een systeem definitie item is een 4-tuple. De opbouw is als volgt :
   - het eerste element is de naam van de file waarin de definitie staat.
   - het tweede element is de naam van de definitie.
   - het derde element is een boolean die aangeeft of de definitie veranderd is.
   - het vierde element is de definitie in een interne weergave. Dit is van het type *Elem Name*.

4. Een tuple definitie is grotendeels gelijk aan die van een systeem definitie. Alleen wordt in het vierde element, van het type *Elem Name*, de tak *Tuple_def* gekozen.

```
type       SysDef = (FileName,SysName,Bool,Elem Name)
type     TupleDef = (FileName,TupleName,Bool,Elem Name)
```

5. Een basis type-defintie wordt opgeslagen in een 3-tuple. De elementen zijn:
   - het eerste element is de naam van de file waarin de basis type-defintie voorkomt
   - het tweede element is de naam van de basis type-defintie.
   - het derde element is van het type *Elem Name* en is de interne weergave van de basis type-defintie. In deze data-constructor wordt de tak *Base* genomen.

6. Een atomaire definitie wordt op dezelfde manier opgeslagen. Alleen het tweede element is nu de naam van de atomaire definitie. En het derde element kiest in de data-constructor *Elem Name* de tak *Atom*.

```
type      BaseDef = (FileName,BaseName,Elem Name)
type      AtomDef = (FileName,AtomName,Elem Name)
```

7. Dit element van de database is de lijst waarin de invoer van de gebruiker bewaard wordt. Hiermee kan de history lijst getoond worden en het repeat commando kan uit deze lijst halen wat er herhaald moet worden. Een element van deze lijst is daarom een string.

```
type     Commandd = String
```

8. Een standaard kosten item is een 2-tuple waarvan het eerste element een string is en het tweede element een integer is. Het eerste element is de naam van het component, het tweede element zijn de kosten van dit component. Deze informatie wordt gebruikt door het commando "bereken de kosten van een systeem", een betekenis functie van de System Semantics.

```
type DefaultCost = (CostName,CostValue)
type     CostName = String
type    CostValue = Int
```

9. Het laatste element in de database is geen lijst maar een 2-tuple. Hierin wordt de laatst ingelezen file bewaard. Het eerste element is de naam van de file, het tweede element is de interne weergave van de file.

```
type     CurSys   = (CurFile,CurValue)
type      CurFile = String
type     CurValue = SSTekst
```

7.2 PROJECTIE FUNCTIES VOOR DE TYPEN SYNONIEMEN VAN DE DATABASE

In de onderstaande tabel 14 staan functies gedefinieerd waarmee de verschillende elementen uit de type synoniemen van de database geselecteerd kunnen worden. Hiermee is het de implementatie van de database weer meer een ADT geworden. Bovendien wordt de code, die met de database werkt, hiermee netter en overzichtelijker.

getEnvName	Env -> EnvName	getEnvName x	fst x
getEnvValue	Env -> EnvValue	getEnvValue x	snd x
getInfoFile	Info -> FileName	getInfoFile x	first3 x
getInfoValue	Info -> InfoValue	getInfoValue x	secon3 x
getInfoChanged	Info -> Bool	getInfoChanged x	third3 x
getSysFile	SysDef -> FileName	getSysFile x	first4 x
getSysName	SysDef -> SysName	getSysName x	secon4 x
getSysChangeValue	SysDef -> Bool	getSysChangeValue x	third4 x
getSystem	SysDef -> Elem Name	getSystem x	fourt4 x
getTupleFile	TupleDef -> FileName	getTupleFile x	first4 x
getTupleName	TupleDef -> TupleName	getTupleName x	secon4 x
getTupleChangeValue	TupleDef -> Bool	getTupleChangeValue x	third4 x
getTuple	TupleDef -> Elem Name	getTuple x	fourt4 x
getPartFile	PartDef -> FileName	getPartFile x	first4 x
getPartName	PartDef -> PartName	getPartName x	secon4 x
getPartChanged	PartDef -> Bool	getPartChanged x	third4 x
getPart	PartDef -> Elem Name	getPart x	fourt4 x
getBaseFile	BaseDef -> FileName	getBaseFile x	first3 x
getBaseName	BaseDef -> BaseName	getBaseName x	secon3 x
getBase	BaseDef -> Elem Name	getBase x	third3 x
getAtomFile	AtomDef -> FileName	getAtomFile x	first3 x
getAtomName	AtomDef -> AtomName	getAtomName x	secon3 x
getAtom	AtomDef -> Elem Name	getAtom x	third3 x
getCostName	DefaultCost -> CostName	getCostName x	fst x
getCostValue	DefaultCost -> CostValue	getCostValue x	snd x
getCurFile	CurSys -> CurFile	getCurFile x	fst x
getCurValue	CurSys -> CurValue	getCurValue x	snd x

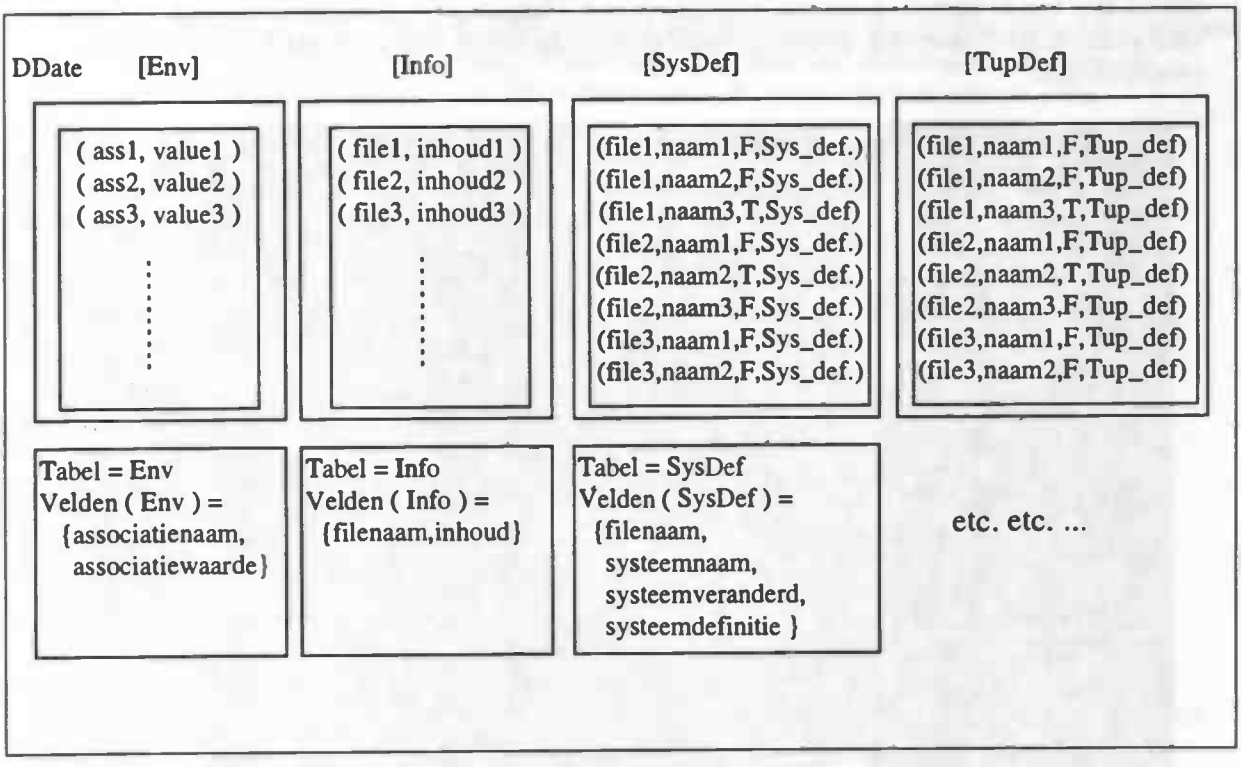
Tabel 14 : Projectie functie voor de type synoniemen van de database.

7.3 ALGEMENE BESPREKING VAN DE DATABASE

De opbouw van de database kan worden vergeleken met die van een relationele database. Ieder element van de data-constructor is dan een tabel. De elementen van ieder element zijn dan de velden. Voor het element `[SysDef]` geeft dit :

- Tabel = SysDef
- Velden (SysDef) = {Filename, SysteemName, Veranderd, SysteemDefinitie}

met enige spontane notatie. Dat de database op deze manier bekeken kan worden is niet vreemd omdat een relationele database eigenlijk ook een functioneel programma is. Het verband tussen een relationele database en de bovenstaande database wordt in figuur 35 grafisch uitgebeeld.



Figuur 35 : Vergelijking met een relationele database.

Hoe met deze database gewerkt wordt zal nu op informele wijze worden besproken. Als een file geladen wordt, dan wordt aan de infolijst een nieuw paar toegevoegd met de naam van de file en de interne weergave. Als de file al geladen is, dan wordt hij niet nog een keer geladen. Verder wordt de file gesplitst in de vier verschillende definities. Deze worden toegevoegd aan de respectievelijke lijsten. Als laatste wordt het element `CurSys` aangepast door ook daarin een paar (filenaam, interne weergave) te plaatsen. Dit element wordt gebruikt voor default keuzen[28].

Wanneer er in een file niet twee definities met dezelfde naam voorkomen wordt iedere definitie uniek bepaald door het paar (filename, definitie naam). Op het dubbel voorkomen van een naam in een file wordt niet gecontroleerd en is nu nog een verantwoordelijkheid van de programmeur.

Als de gebruiker een nieuwe commando invoert dan wordt deze regel toegevoegd aan het element `Commandd`. Dit is de lijst van alle ingevoerde commando's. Bij toevoegen van een nieuwe regel wordt de lengte van de lijst beperkt tot de waarde van globale expressie `lenhistory`.

De environment kan worden geraadpleegd en veranderd met het commando "set"[29]. Dit is niet moeilijk. De verzameling standaardkosten kan worden veranderd en geraadpleegd via het commando "kosten[30]". Ook dit is niet moeilijk.

---

[28] Bij een aantal opdrachten in de commandoshell moet een filenaam opgegeven worden. Als die niet wordt opgegeven dan gaat de commandoshell uit van de file uit `CurSys`. Dus de laatst ingelezen file.

[29] Dit is de optie "s" in de commandoshell

[30] Dit is de optie "c" in de commandoshell

7.4.1 Inleiding

De functies waarmee met de database gewerkt kan worden staan in de file **comproj.hs**. Deze file is erg groot en wordt daarom niet compleet besproken. Een aantal interessante functies en feiten zijn er uitgelicht.

Bij veel database functies wordt gebruik gemaakt van de volgende twee functies, die beiden in de module **ssutil.hs** staan :

```
present :: ( a -> Bool ) -> [a] -> Bool
```

Deze functie bepaalt of in een lijst van het type a er een element is dat voldoet aan het predikaat.

```
find    :: (a -> Bool) -> [a] -> a
```

Deze functie zoekt in een lijst van het type a naar het eerste element dat voldoet aan het predikaat.

Voor de functie *find* moet het zeker zijn dat er zo'n element in de lijst is. Anders gaat het fout. Daarom worden deze twee functies vaak in combinatie gebruikt. Eerst wordt met *present* gekeken of het element er is en zo ja, dan wordt dit element er met *find* uitgehaald.

Dit heeft ook gevolgen voor de functies die gebruik maken van *present* en *find*. Ook deze functies moeten beiden gebruikt worden. De volgende functie kan bijvoorbeeld niet alleen worden gebruikt.

```
getEnvItem :: MR DesignData -> EnvName -> IO (Env)
getEnvItem designDB envname =
        getEnvList designDB >>= \xs ->
        return (find (\x -> getEnvName x == envname ) xs)
```

Stel dat er geen element is waarvan de naam gelijk is aan het argument. Dan gaat deze functie in de fout. Daarom moet met een andere functie eerst gekeken worden of er zo'n element is. Dit kan met de volgende functie :

```
isEnvItemLoaded :: MR DesignData -> EnvName -> IO(Bool)
isEnvItemLoaded designDB envname =
          getEnvList designDB >>= \envlist ->
          return (present (\x -> getEnvName x == envname ) envlist)
```

Een stuk programma code zal dan zijn :

```
isEnvItemLoaded ..... >>= \value ->
if value then getEnvItem .....
```

Deze combinatie komt veel voor wanneer er een element uit een lijst gehaald moet worden. Omdat het goed fout kan gaan is het zeer belangrijk hieraan te denken. Daarvan hier akte.

Een andere opmerking betreft de argumenten. De naamgeving van de argumenten is volgens de conventie van § 5.2. Een filename is van het type FileName enzovoort. Verder is het eerste argument bij bijna iedere functie van het type :

```
MR DesignData -> .......
```

Dit eerste argument is de database, verpakt in een State Transformer. Hiervoor is gekozen omdat bleek dat de code, die gebruik maakt van de database functies, hierdoor overzichtelijker werd. Er had ook gekozen kunnen worden voor :

```
DesignData -> ....
```

maar dan moest in die code steeds gebruik gemaakt worden van de volgende constructie :

```
readVar designdata >>= \ddata -> .....
```

en dat is minder overzichtelijk.

In de code die gebruik maakt van de database functies hoeft nu helemaal geen rekening gehouden te worden met de verpakking van de database in een State Transformer. Dit gebeurt wel in de database functies. De database is gewoon een argument, meer niet. Door deze keuze is het laatste argument ook een State Transformer.

De database functies zijn op te delen in een aantal categorieën :

- geef een complete verzameling elementen.
  In termen van een relationele database : Geef een tabel
- geef een element.
  In termen van een relationele database : **Select** een record gegeven een zoekterm
- geef een aantal elementen
  In termen van een relationele database : **Select** een aantal records gegeven een zoekterm
- voeg een element toe.
  In termen van een relationele database : **Append** een record
- voeg een aantal elementen toe.
  In termen van een relationele database : **Append** meerdere records
- verwijder een element
  In termen van een relationele database : **Delete** een record
- is een element in de database aanwezig
  In termen van een relationele database : **Find** een record gegeven een zoekterm

Bijna iedere element, anders gezegd tabel, in de database heeft deze functies. Toegevoegd zijn functies die informatie gegeven over de status van de elementen. Zo kan van sommige elementen bepaald worden of ze veranderd zijn na het inlezen. Dit kan gebruikt worden bij bijvoorbeeld het beëindigen van het programma waarmee gevraagd kan worden of de veranderde systemen opgeslagen moeten worden.
Een voorbeeld van deze opdeling in categorieën zijn de functies voor het element Env[31] .

[31] Met de relationele veldnamen uit figuur 30.

getEnvList db	Haal de environment uit de database : **Geef** de tabel
getEnvItem db x	Haal uit de environment het paar met de associatie naam x **Select** het element met (associatienaam = x)
addEnvList db lijst	Voeg aan de environment de lijst toe. **Append** een lijst met records
addEnvItem db (x,y)	Voeg aan de environment het paar toe. **Append** een record
remEnvItem db x	Verwijder uit de environment het element met de naam x **Delete** record met (associatienaam = x)
isEnvItemLoaded db x	Kijkt of er in de environment een associatie paar is met de naam x **Find** record met (associatienaam = x)

Tabel 15 : De verschillende categorieën database functies toegepast op het database onderdeel *Env*.

Uit de verzameling functies worden nu een aantal interessante en / of representatieve functies besproken.

```
addEnvList :: MR DesignData -> [Env] -> IO ()
addEnvList designDB envlist = readVar designDB >>== \tempDB ->
        add tempDB envlist
  where add (DData a b c d e f g h i) denvlist =
          designDB >>:: DData (denvlist ++ a) b c d e f g h i

addEnvItem :: MR DesignData -> Env -> IO ()
addEnvItem designDB envitem = readVar designDB >>== \tempDB ->
        add tempDB envitem
  where add (DData a b c d e f g h i) envitem =
          designDB >>:: DData (envitem:a) b c d e f g h i
```

Deze twee functies zijn als voorbeeld genomen voor het toevoegen van een (lijst) element(en). Met *readVar* wordt "de inhoud" van de database State Transformer in *tempDB* opgevraagd. Via de hulp functie *add* wordt de structuur van de data-constructor blootgelegd en kan op de juiste plaats worden toegevoegd.

Er wordt niet gecontroleerd op dubbele voorkomens om de database functies simpel en algemeen toepasbaar te houden. De programmeur moet zelf controleren op het dubbel voorkomen van een element. Misschien dat later toch nog zo'n functie "standaard" gemaakt gaat worden.

```
getEnvList :: MR DesignData -> IO ([Env])
getEnvList designDB = readVar designDB >>== \tempenv ->
      return (get tempenv)
 where get (DData x _ _ _ _ _ _ _ _) = x

getEnvItem :: MR DesignData -> EnvName -> IO (Env)
getEnvItem designDB envname =
        getEnvList designDB >>= \xs ->
        return (find (\x -> getEnvName x == envname ) xs)
```

Deze twee functies zijn als voorbeeld genomen voor het opvragen van een (lijst) element(en). Met *getEnvList* wordt het hele element, tabel, opgehaald. Deze functie is gemakkelijk. Met *getEnvItem* wordt één element, record, opgevraagd. Met de functie *getEnvList* wordt de juiste tabel opgehaald en met de functie *find* wordt deze tabel doorlopen om het eerste element op te vragen waarvan de associatie naam gelijk is aan het tweede argument. In deze laatste functie is goed te zien hoe gewerkt wordt met de projectie functies uit § 7.2 bij de expressie :

```
getEnvName x == envname.
```

De functie getEnvItem moet samen gebruikt worden met isEnvItemLoaded vanwege het gebruik van *find*.

```
remEnvItem :: MR DesignData -> EnvName -> IO ()
remEnvItem designDB envitem = readVar designDB >>== \tempDB ->
        rem tempDB envitem
  where rem (DData a b c d e f g h i) envitem =
          designDB >>:: DData (ritem a envitem) b c d e f g h i
        ritem xs name = filter ( \x -> getEnvName x /= name ) xs
```

De functie *remEnvItem* verwijderd een element uit de tabel *Env*. De functie is niet moeilijk en representatief voor alle functies die elementen verwijderen. Met *readVar* wordt "de inhoud" van de database State Transformer in *tempDB* opgevraagd. Via de hulp functie rem wordt de structuur van de data-constructor blootgelegd en kan op de juiste plaats worden verwijderd. Met de *filter* functie wordt de lijst *tempDB* doorlopen waarbij alle elementen waarvan de associatie naam niet gelijk is aan het tweede argument bewaard blijven.

```
isEnvItemLoaded :: MR DesignData -> EnvName -> IO(Bool)
isEnvItemLoaded designDB envname =
          getEnvList designDB >>= \envlist ->
          return (present (\x -> getEnvName x == envname ) envlist)
```

Deze functie bepaald met behulp van de functie present of een element aanwezig is. De functie is na uitleg van de vorige functies gemakkelijk.

```
getSysDefListFile :: MR DesignData -> FileName -> IO [SysDef]
getSysDefListFile designDB filename =
        getSysDefList designDB >>= \xs ->
        return (filter (\elem -> getSysFile elem == filename ) xs)

getSysDefListName :: MR DesignData -> SysName -> IO [SysDef]
getSysDefListName designDB sysname =
        getSysDefList designDB >>= \xs ->
        return (filter (\elem -> getSysName elem == sysname ) xs)

getSysDefItemFileName :: MR DesignData -> FileName -> SysName -> IO
SysDef
getSysDefItemFileName designDB fname sname =
        getSysDefList designDB >>= \xs ->
        return (find (\x -> getSysName x == sname &&
                            getSysFile x == fname
                    ) xs)
```

Iedere definitie is uniek te bepalen door het paar : (filenaam, definitie naam). De bovenstaande functies kunnen definities uit de database halen waarbij het zoeken bepaald wordt door :

- alleen een filenaam
- alleen een definitie naam
- een filename en een definitie naam

De eerste functies haalt alle definities van een ingelezen file. De tweede haalt alle definities uit de database met dezelfde definitie naam. De derde haalt één definitie uit de database waarbij de definitie uit file x komt en de naam y moet hebben. Bij de verschillende functies wordt de database op verschillende manieren doorzocht. Hoe dat gebeurt blijkt uit de naamgeving. Alleen "**File**" in de naam van de functie geeft optie 1. Alleen "**Name**" in de naam van de functie geeft optie 2. Zowel "**File**" en "**Name**" geeft optie 3. Voor de vier verschillende soorten definities zijn dergelijke functies gedefinieerd.

Met deze inleiding moet het niet moeilijk zijn om de functies te begrijpen en te gebruiken. In de volgende paragrafen wordt voor ieder van de elementen van de database de functies besproken.

7.4.2 Functies van het database onderdeel Env

Typering van de functies voor het database onderdeel Env	
getEnvList	MR DesignData -> IO ([Env])
getEnvItem	MR DesignData -> EnvName -> IO (Env)
addEnvList	MR DesignData -> [Env] -> IO ()
addEnvItem	MR DesignData -> Env -> IO ()
remEnvItem	MR DesignData -> EnvName -> IO ()
isEnvItemLoaded	MR DesignData -> EnvName -> IO(Bool)

Tabel 16 : Typering van de functies voor het database onderdeel *Env*.

Gebruik en betekenis van de functies voor het database onderdeel Env	
getEnvList **db**	Haalt uit de database **db** de environment
getEnvItem **db assoc-name**	Haalt uit de database **db** het associatie paar **assoc-name**
addEnvList **db assoc-list**	Voegt aan de database **db assoc-list** toe. Dubbelen kunnen voorkomen.
addEnvItem **db assoc-pair**	Voegt aan de database **db assoc-pair** toe. Dubbelen kunnen voorkomen.
remEnvItem **db assoc-name**	Verwijderd uit de database **db** het associatie paar **assoc-name**
isEnvItemLoaded **db assoc-name**	Bepaalt of in de database **db** het associatie paar **assoc-name** voorkomt

Tabel 17: Gebruik en betekenis van de functies voor het database onderdeel *Env*.

7.4.3 Functies van het database onderdeel Info

Typering van de functies voor het database onderdeel Info	
getInfoList	MR DesignData -> IO ([Info])
getInfoItem	MR DesignData -> FileName -> IO (SSTekst)
addInfoList	MR DesignData -> [Info] -> IO ()
addInfoItem	MR DesignData -> Info -> IO ()
isInfoItemLoaded	MR DesignData -> FileName -> IO (Bool)
isInfoItemChanged	MR DesignData -> FileName -> IO (Bool)
remInfoItem	MR DesignData -> FileName -> IO ()

Tabel 18 : Typering van de functies voor het database onderdeel *Info*.

Gebruik en betekenis van de functies voor het database onderdeel Info

getInfoList **db**	Haalt uit de database **db** de lijst met alle files
getInfoItem **db file-name**	Haalt uit de database **db** het info-item **file-name**
addInfoList **db info-list**	Voegt aan de database **db info-list** toe. Er wordt niet gecontroleerd op dubbele voorkomens.
addInfoItem **db info-name**	Voegt aan de database **db info-name** toe. Er wordt niet gecontroleerd of infoitem al in de database voorkomt.
isInfoItemLoaded **db file-name**	Bepaalt of in de database **db file-name** voorkomt.
isInfoItemChanged **db file-name**	Bepaalt of **file-name** uit database **db** veranderd is ten opzichte van het moment van inlezen. Dit wordt gecontroleerd door te bepalen of de tot de file behoorende systeem en / of tuple definities veranderd zijn.
remInfoItem **db file-name**	Verwijdert **file-name** uit de database **db**.

Tabel 19 : Gebruik en betekenis van de functies voor het database onderdeel *Info* .

7.4.4 Functies van het database onderdeel SysDef

Typering van de functies voor het database onderdeel SysDef

getSysDefList	MR DesignData -> IO [SysDef]
getSysDefListFile	MR DesignData -> FileName -> IO [SysDef]
getSysDefListName	MR DesignData -> SysName -> IO [SysDef]
getSysDefItemFileName	MR DesignData -> FileName -> SysName -> IO SysDef
addSysDefList	MR DesignData -> [SysDef] -> IO ()
addSysDefItem	MR DesignData -> SysDef -> IO ()
isSysDefChanged	[SysDef] -> FileName -> Bool
setTupleDefName	MR DesignData -> FileName -> FileName -> IO()
remSysDefItemFileName	MR DesignData -> FileName -> SysName -> IO ()
remSysDefListFile	MR DesignData -> FileName -> IO ()
isSysDefItemLoadedFileName	MR DesignData -> FileName -> SysName -> IO Bool
isSysDefItemLoadedName	MR DesignData -> SysName -> IO Bool

Tabel 20 : Typering van de functies voor het database onderdeel *SysDef*.

Gebruik en betekenis van de functies voor het database onderdeel Info

getSysDefList **db**	Haalt uit de database **db** alle systeem definities
getSysDefListFile **db file-name**	Haalt uit de database **db** alle systeem definities van **file-name**
getSysDefListName **db system-name**	Haalt uit de database **db** alle systeem definities met **system-name**
getSysDefItemFileName **db file-name system-name**	Haalt uit de database **db** één systeem definitie uit **file-name** en met definitie naam **system-name**
addSysDefList **db system-list**	Voegt aan de database **db** de lijst **system-list** toe
addSysDefItem **db system-name**	Voegt aan de database **db** element **system-name** toe.
isSysDefChanged **system-list file-name**	Bepaalt of een systeem definitie uit **system-list** uit **file-name** is veranderd
setSysDefName**db file-name file-name-2**	Verandert **file-name** in **file-name-2** van alle systeemdefities in **db** uit file-name
remSysDefItemFileName **db file-name system-name**	Verwijdert uit de database **db** de systeem definitie uit **file-name** en met definitie naam **system-name**
remSysDefListFile **db file-name**	Verwijdert uit de database **db** de systeem definities uit **file-name**
isSysDefItemLoadedFileName **db file-name system-name**	Bepaalt of systeem definitie uit **file-name** en met definitie naam **system-name** in **db** geladen is
isSysDefItemLoadedName **db system-name**	Bepaalt of systeem definitie met definitie naam **system-name** in **db** geladen is

Tabel 21 : Gebruik en betekenis van de functies voor het database onderdeel *SysDef*.

7.4.5 Functies van het database onderdeel TupleDef

Typering van de functies voor het database onderdeel TupleDef

getTupleDefList	MR DesignData -> IO [TupleDef]
getTupleDefListFile	MR DesignData -> FileName -> IO [TupleDef]
getTupleDefListName	MR DesignData -> TupleName -> IO [TupleDef]
getTupleDefItemFileName	MR DesignData -> FileName -> TupleName -> IO TupleDef
addTupleDefList	MR DesignData -> [TupleDef] -> IO ()
addTupleDefItem	MR DesignData -> TupleDef -> IO ()
isTupleDefChanged	[TupleDef] -> FileName -> Bool

7.4.6 Functies van het database onderdeel BaseDef

Typering van de functies voor het database onderdeel BaseDef

getBaseDefList	MR DesignData -> IO [BaseDef]
getBaseDefListFile	MR DesignData -> FileName -> IO [BaseDef]
getBaseDefListName	MR DesignData -> BaseName -> IO [BaseDef]
getBaseDefItemFileName	MR DesignData -> FileName -> BaseName -> IO BaseDef
addBaseDefItem	MR DesignData -> BaseDef -> IO ()
addBaseDefList	MR DesignData -> [BaseDef] -> IO ()
setBaseDefName	MR DesignData -> FileName -> FileName -> IO()
remBaseDefListFile	MR DesignData -> FileName -> IO ()
remBaseDefItemFileName	MR DesignData -> FileName -> BaseName -> IO ()
isBaseDefItemLoadedName	MR DesignData -> BaseName -> IO Bool
isBaseDefItemLoadedFileName	MR DesignData -> FileName -> BaseName -> IO Bool

Tabel 24 : Typering van de functies voor het database onderdeel *BaseDef*.

Gebruik en betekenis van de functies voor het database onderdeel BaseDef

getBaseDefList **db**	Haalt uit de database **db** alle basis type-definties
getBaseDefListFile **db file-name**	Haalt uit de database **db** alle basis type-definties van **file-name**
getBaseDefListName **db base-name**	Haalt uit de database **db** alle basis type-definties met **base-name**
getBaseDefItemFileName **db file-name base-name**	Haalt uit de database **db** één basis type-defintie uit **file-name** en met definitie naam **base-name**
addBaseDefList **db base-list**	Voegt aan de database **db** de lijst **base-list** toe
addBaseDefItem **db base-name**	Voegt aan de database **db** element **base-name** toe.
setBaseDefName **db file-name file-name2**	Verander **file-name** in **file-name-2** van alle basis type defities in **db** uit **file-name**
remBaseDefItemFileName **db file-name base-name**	Verwijdert uit de database **db** de basis type-defintie uit **file-name** en met definitie naam **base-name**
remBaseDefListFile **db file-name**	Verwijdert uit de database **db** de basis type-definties uit **file-name**
isBaseDefItemLoadedFileName **db file-name base-name**	Bepaalt of basis type-defintie uit **file-name** en met definitie naam **base-name** in database **db** geladen is

isBaseDefItemLoadedName **db base-name**	Bepaalt of basis type-defintie met definitie naam **base-name** in database **db** geladen is

Tabel 25 : Gebruik en betekenis van de functies voor het database onderdeel *BaseDef*.

7.4.7 Functies van het database onderdeel AtomDef

Typering van de functies voor het database onderdeel AtomDef

getAtomDefList	MR DesignData -> IO [AtomDef]
getAtomDefListFile	MR DesignData -> FileName -> IO [AtomDef]
getAtomDefListName	MR DesignData -> AtomName -> IO [AtomDef]
getAtomDefItemFileName	MR DesignData -> FileName -> AtomName -> IO AtomDef
addAtomDefItem	MR DesignData -> AtomDef -> IO ()
addAtomDefList	MR DesignData -> [AtomDef] -> IO ()
setAtomDefName	MR DesignData -> FileName -> FileName -> IO()
remAtomDefListFile	MR DesignData -> FileName -> IO ()
remAtomDefItemFileName	MR DesignData -> FileName -> AtomName -> IO ()
isAtomDefItemLoadedName	MR DesignData -> AtomName -> IO Bool
isAtomDefItemLoadedFileName	MR DesignData -> FileName -> AtomName -> IO Bool

Tabel 26 : Typering van de functies voor het database onderdeel *AtomDef*.

Gebruik en betekenis van de functies voor het database onderdeel AtomDef

getAtomDefList **db**	Haalt uit de database **db** alle atomaire type-defintie
getAtomDefListFile **db file-name**	Haalt uit de database **db** alle atomaire type-defintie van **file-name**
getAtomDefListName **db atom-name**	Haalt uit de database **db** alle atomaire type-defintie met **atom-name**
getAtomDefItemFileName **db file-name atom-name**	Haalt uit de database **db** één atomaire type-defintie uit **file-name** en met definitie naam **atom-name**
addAtomDefList **db atom-list**	Voegt aan de database **db** de lijst **atom-list** toe
addAtomDefItem **db atom-name**	Voegt aan de database **db** element **atom-name** toe.
setAtomDefName **db file-name file-name-2**	Verander file-name in **file-name-2** van alle atomaire defities in **d** uit **file-name**
remAtomDefItemFileName **db file-name atom-name**	Verwijdert uit de database **db** de atomaire type-defintie uit **file-name** en met definitie naam **atom-name**

remAtomDefListFile **db file-name**	Verwijdert uit de database **db** de atomaire type-definties uit **file-name**
isAtomDefItemLoadedFileName **db file-name atom-name**	Bepaalt of atomaire type-defintie uit **file-name** en met definitie naam **atom-name** in database **db** geladen is
isAtomDefItemLoadedName **db atom-name**	Bepaalt of atomaire type-defintie met definitie naam **atom-name** database **db** geladen is

Tabel 27 : Gebruik en betekenis van de functies voor het database onderdeel *AtomDef*.

7.4.8 Functies van het database onderdeel Commandd

Typering van de functies voor het database onderdeel Commandd

getCommandList	MR DesignData -> IO [Commandd]
getCurrentCommand	MR DesignData -> IO Commandd
getLastCommand	MR DesignData -> IO Commandd
getCommandIndex	MR DesignData -> Int -> IO Commandd
addCommandList	MR DesignData -> [Commandd] -> IO ()
addCommandItem	MR DesignData -> Commandd -> IO ()

Tabel 28 : Typering van de functies voor het database onderdeel *Commandd*

Gebruik en betekenis van de functies voor het database onderdeel Commandd

getCommandList db	Haalt uit de database db de commando lijst
getCurrentCommand db	Haalt uit de database db het huidige commando
getLastCommand db	Haalt uit de database db het laatste commando
getCommandIndex db number	Haalt uit de database db het i-de commando
addCommandList db command-list	Voegt aan de database db de lijst command-list toe
addCommandItem db command	Voegt aan de database db element command toe

Tabel 29 : Gebruik en betekenis van de functies voor het database onderdeel *Commandd*

7.4.9 Functies van het database onderdeel DefaultCost

Typering van de functies voor het database onderdeel DefaultCost

getCostList	MR DesignData -> IO [DefaultCost]
getCostItem	MR DesignData -> CostName -> IO Int
addCostList	MR DesignData -> [ECostpair] -> IO ()
addCostItem	MR DesignData -> ECostpair -> IO ()
isCostItemLoaded	MR DesignData -> CostName -> IO Bool
remCostItem	MR DesignData -> CostName -> IO ()

Tabel 30 : Typering van de functies voor het database onderdeel `DefaultCost`.

Gebruik en betekenis van de functies voor het database onderdeel DefaultCost

getCostList **db**	Haalt uit de database **db** de standaard kosten lijst
getCostItem **db cost-name**	Haalt uit de database **db** de standaard kosten definitie van **cost-name**
addCostList **db cost-list**	Voegt aan de database **db** de lijst **cost-list** toe
addCostItem **db cost-item**	Voegt aan de database **db** element **cost-item** toe
isCostItemLoaded **db cost-name**	Bepaalt of standaard kosten **cost-name** in database **db** geladen is
remCostItem **db cost-name**	Verwijdert uit de database **db** de standaard kosten **cost-name**

Tabel 31 : Gebruik en betekenis van de functies voor het database onderdeel `DefaultCost`

7.4.10 Functies van het database onderdeel CurSys

Typering van de functies voor het database onderdeel CurSys

addCurSys	MR DesignData -> CurFile -> CurValue -> IO ()
getCurSys	MR DesignData -> IO CurSys
getCurFileName	MR DesignData -> IO FileName
isCurSysLoaded	MR DesignData -> IO Bool
isPartLoadedCurSys	MR DesignData -> PartName -> IO Bool
getPartCurSys	MR DesignData -> PartName -> IO (Elem Name)
isSystemInCurSys	MR DesignData -> SysName -> IO Bool
getSystemInCurSys	MR DesignData -> SysName -> IO (Elem Name)
isTupleInCurSys	MR DesignData -> TupleName -> IO Bool
getTupleInCurSys	MR DesignData -> TupleName -> IO (Elem Name)

Tabel 32 : Typering van de functies voor het database onderdeel `CurSys`.

Gebruik en betekenis van de functies voor het database onderdeel CurSys

addCurSys **db curfile-name cur-value**	Voegt aan de database **db** het paar **(curfile-name,cur-value)** toe
getCurSys **db**	Haalt uit de database **db** het laatst ingelezen systeem
getCurFileName **db**	Haalt uit de database **db** de naam van de laatst ingelezen file
isCurSysLoaded **db**	Bepaalt of in database **db** al een file ingelezen is
isPartLoadedCurSys **db part-name**	Bepaalt of in de database **db** een onderdeel met de naam **part-name** geladen is.
getPartCurSys **db part-name**	Haalt uit de database **db** een onderdeel met de naam **part-name**
isSystemInCurSys **db system-name**	Bepaalt of in database **db system-name** in het huidige file voorkomt
getSystemInCurSys **db system-name**	Haalt uit de database **db** een systeem met de naam **system-name**
isTupleInCurSys **db tuple-name**	Bepaalt of in database **db tuple-name** in het huidige file voorkomt
getTupleInCurSys **db tuple-name**	Haalt uit de database **db** een tuple met de naam **tuple-name**

Tabel 33 : Gebruik en betekenis van de functies voor het database onderdeel `CurSys`.

7.5 DE MODULE COMMHGDB.HS

In deze module staan een aantal "queries". In tegenstelling tot de functie uit **commproj.hs** zijn dit functies die functies uit **commproj.hs** samenstellen tot grotere functies. De functies staan in tabel 34.

- ```
  mkInitDataBase :: IO DesignData
  mkInitDataBase =
          make_cost_list          >>= \costresult ->
          make_env_list           >>= \envresult ->
          return (DData envresult [][][][][][][] costresult ("",[]))
    where make_cost_list =
               readAFile "commonct.lst" >>= \ioTuple ->
               case (fst ioTuple) of
               True  -> return
                           (mkcostlist (startArglex (snd ioTuple)) [])
               False -> writeln (snd ioTuple) >>
                        return []
               where mkcostlist [] accum          =
                                  accum
                     mkcostlist (x:(y:xs)) accum =
                                  mkcostlist xs ((x,stringtoint y):accum)
          make_env_list =
               readAFile "setting.lst" >>= \ioTuple ->
               case (fst ioTuple) of
               True  -> return
                           (mkenvlist(startSetlex (snd ioTuple)) [])
               False -> writeln (snd ioTuple) >>
                        return []
               where mkenvlist [] accum          =
                                  accum
                     mkenvlist (x:(y:xs)) accum =
                                  mkenvlist xs ((x,y):accum)
  ```

Deze functie wordt gebruikt voor het maken van een lege database. Bij het initialiseren van het programma wordt de functie *mkInitDataBase* aangeroepen. De functies *make_cost_list* en *make_env_list* proberen het element *DefaultCost* respectievelijk *Env* te initialiseren door het inlezen van de files **commonct.lst** en **setting.lst**. Het is op een soortgelijke manier denkbaar dat een prelude file wordt ingelezen met een aantal standaard definities[32].

De functies *..Part..* worden gebruikt wanneer niet specifiek een systeem of tuple wórden gezocht. In deze functies wordt eerst gekeken of er een systeem is dat voldoet aan de eisen. Is dit zo dan wordt dit systeem of *True* opgeleverd. Is dit niet zo dan wordt gekeken of er een tuple is dat aan de eisen voldoet. Is dit zo, dan wordt dit tuple of true opgeleverd. Is dit niet zo dan wordt *False* opgeleverd.

Met de functie *getAllFile* worden alle definities van één file uit de database gehaald. Hiervoor worden de basisfuncties :

- *getSysDefListFile*
- *getTupleDefListFile*
- *getBaseDefListFile*
- *getAtomDefListFile*

De functie *savefile* wordt gebruikt om een file te bewaren. Er moeten twee filenamen als argument meegegeven worden waarvan de tweede eventueel leeg kan zijn. Is het tweede argument een lege string dan wordt de file met dezelfde naam bewaard. Anders wordt de file onder de andere naam bewaard.

---

[32] Vergelijkbaar met bijvoorbeeld de **cc.prelude** file van Gofer
  ```

mkInitDataBase	IO DesignData
isPartItemLoadedFileName	MR DesignData -> FileName -> PartName -> IO Bool
isPartItemLoadedName	MR DesignData -> PartName -> IO Bool
getPartItemName	MR DesignData -> PartName -> IO(FileName,PartName,Bool,Elem Name)
getPartList	MR DesignData -> IO [PartDef]
getPartFileName	MR DesignData -> FileName -> PartName -> (IO FileName, PartName, Bool, Elem Name)
getAllFile	MR DesignData -> FileName -> IO[Elem Name]
remove_file	MR DesignData -> FileName -> IO()
changeFileName	MR DesignData -> FileName -> FileName -> IO()
savefile	MR DesignData -> FileName -> FileName -> IO()
isAnyInfoItemChanged	MR DesignData -> IO Bool
getChangedInfoList	MR DesignData -> IO [FileName]

Tabel 34 : De functies uit de file **commhgdb.hs**.

De functie *remove_file* verwijderd een file uit de database. Hiervoor worden de basisfuncties

- *remInfoItem*
- *remSysDefListFile*
- *remTupleDefListFile*
- *remBaseDefListFile*
- *remAtomDefListFile*

gebruikt.

De functie *changeFileName* veranderd de naam van een file in alle definities in een andere naam. De functie *isAnyInfoItemChanged* bepaalt of ergens in de database een definitie veranderd is. Deze functie wordt onder andere bij het afsluiten van het programma gebruikt. Met de functie *getChangedInfoList* kan deze lijst van veranderde definities opgehaald worden.

8 DE HELP MODULE

Bij een aantal commando's uit de commando set kan hulp worden gevraagd over de syntax en / of betekenis van het commando. Sommige commando's kennen geen help omdat zij nog niet uitgewerkt zijn. In de loop van de ontwikkeling van de ontwikkelomgeving moeten deze gaten dus gevuld worden.

De module met de help commando's is niet moeilijk. Als de functies zo gemakkelijk zijn waarom dan toch een aparte module? Hier is voor gekozen omdat op deze manier een goed overzicht te houden is op het uiterlijk van de help berichten. En verder omdat een goede help file in mijn ogen belangrijk is. Als hierdoor de boeken achterwege kunnen blijven, des te beter. Niet dat deze helpfile nu direct een oplossing daarvoor zal zijn. Maar door de betere beheersbaarheid ten gevolge van de modulaire aanpak zijn de kansen op succes bij een boomsparende actie groter.

In onderstaande tabel staan de help functies op een rij :

helpcom	help commando dat aangeeft wat de commando's zijn van de shell	
behavehelp	help commando voor het gedrags commando	niet ingevuld
costhelp	help commando voor het berekenen van kosten	uitgebreid
file_designhelp	help commando voor het ontwerpen met een file	simpel
file_edithelp	help commando voor bewerken van een file.	niet ingevuld
grafisch_designhelp	help commando voor het grafisch ontwerpen	simpel
grafisch_edithelp	help commando voor het grafisch aanpassen	niet ingevuld
helphelp	help commando voor het help commando	simpel
historyhelp	help commando voor history command	uitgebreid
loadhelp	help commando voor het laden van systemen	uitgebreid
patternhelp	help commando voor het maken van testpatronen	uitgebreid
prettyadhelp	help commando voor afdrukken van interne weergave	uitgebreid
prettyhelp	help commando voor afdrukken van programma tekst weergave	uitgebreid
quithelp	help commando voor quit commando	uitgebreid
removehelp	help commando voor het verwijderen van systemen	uitgebreid
repeathelp	help commando voor repeat command	uitgebreid
savehelp	help commando voor het bewaren van systemen	uitgebreid
sethelp	help commando voor set commando	uitgebreid
shellhelp	help commando voor het shell commando	uitgebreid
showstatehelp	help commando voor het tonen van de database	uitgebreld
statichelp	help commando voor het tonen van statistieken	simpel
transform1help	help commando voor een transformatie commando	niet ingevuld
xilinxshelp	help commando voor produceren van Xilinx uitvoer	uitgebreid

Tabel 35 : Overzicht van de help functies. Niet ingevuld = commando is nog niet aanwezig.
Simpel = extra hulp is niet aanwezig.
Uitgebreid = extra hulp is aanwezig.

9 UTILITIES EN COMPATIBILITEIT

Voor de commandoshell zijn 2 modules gemaakt met utitlities :

- **ssutil.hs**
- **commutil.hs**

Verder zijn er twee modules gemaakt voor de compatibiliteit tussen Gofer en de GHC-compiler. Ondanks dat dit soms wel veel werk gaf, is toch gekozen voor het handhaven van de uitwisselbaarheid van de code.

Vooral in het testen is Gofer veel handiger dan de GHC-compiler. Snel even iets proberen in Gofer is geen probleem. Snel even iets proberen met de GHC-compiler is langzaam iets proberen. Bij Gofer krijg je direct de foutmelding om je oren, bij de GHC-compiler moet hier lang, lang, lang op gewacht worden. Het is dan ook aan te raden aan de opvolgers om deze compatibiliteit door middel van deze twee modules te handhaven.

Het maken van deze modules is ook een bewijs van de kracht van functioneel programmeren. Als er een bepaalde functie niet is, dan maak je gewoon een dummy-functie van het juiste type, en de code loopt weer. Een voorbeeld is de functie writeFile uit de GHC-compiler. Deze functie kent Gofer niet een IO() vorm. Maar die kun je gewoon zelf maken.

De twee files zijn :

- **compgof.hs**
- **compghc.hs**

9.1 COMPATIBILITEIT

De beide files zijn te verdelen in een aantal stukken :

1. Functies[33] die wel in Gofer voorkomen maar niet in de GHC-compiler. Deze functies staan in file **compghc.hs**.
2. Functies die wel in de GHC-compiler voorkomen maar niet in Gofer. Deze functies staan in file **compgof.hs**. Indien mogelijk worden deze functie zo gedefinieerd dat ze "semantisch gesproken" hetzelfde doen. Is dit niet mogelijk[34] dan wordt alleen het type geimiteerd dat de functie uit de GHC-compiler oplevert.
3. Overeenkomende functies, qua "semantiek" en typering, die in beiden[35] voorkomen maar een andere naam hebben. Bij deze functies is steeds gekozen voor het herbenoemen van de Gofer functies naar de naam van de functies uit de GHC-compiler. Deze functies staan in file **compgof.hs**.
4. Functies die in beiden voorkomen, met dezelfde "semantiek", maar die niet overeenkomen omdat de typering verschilt. In deze gevallen is ervoor gekozen om een nieuwe "verse" functie in beide files te definieren en de typering van de GHC-functies te imiteren in de Gofer compatibiliteits file.

   In de nieuwe functie wordt in de GHC-file de naam van de conflicterende functie gezet. Dit is dus alleen maar herbenoemen van de conflicterende functie.

   In de Gofer file wordt het resultaat geimiteerd. In Gofer hebben deze functies dus geen zin, maar hiermee blijft de code wel compatibal.

---

[33] Hier moet gelezen worden : functies en datatypen.

[34] Bijvoorbeeld in het geval van `compExistFile`. Er is geen functie in Gofer die hetzelfde doet en die functie is ook niet te maken. Daarom wordt een expressie van het gewenste type opgelevert.

[35] In Gofer en de GHC-compiler.

9.1.1 Compghc.hs

De functies uit de eerste groep zijn :

fst3	(a,b,c) -> a
snd3	(a,b,c) -> b
thd3	(a,b,c) -> c
copy	Int -> a -> [a]
space	Int -> String

Deze algemene projectie-functies en utilities komen wel in de prelude voor van Gofer, maar niet in die van de GHC-compiler. Het is natuurlijk mogelijk om in iedere file, waar dit nodig is, deze functies opnieuw te definieren. En vanwege een vast wel geldige reden is het in sommige files ook wel gedaan, maar dit druist in tegen de regels van modulair werken. Een andere, nette oplossing, is het aanpassen van de prelude file van de GHC-compiler zodat deze functie wel standaard zijn. Maar over deze file heb ik thuis niet de beschikking vanwege een *"ready-to-run-and-not-to-be-modified"* versie van de compiler.

De functies uit de vierde groep zijn :

type MR x	MutableVar _RealWorld x
type TestPattern	MA Int Bool
type MA b c	_MutableArray _RealWorld b c
readAAFile	String -> IO String
writeAAFile	String -> String -> IO ()
getProgramName	IO String
getArguments	IO [String]
getEnvSetting	String -> IO String

Van deze functies komen in de file `compgof.hs` spiegelbeelden voor die in ieder geval van het zelfde type zijn. En indien mogelijk doen ze ook nog hetzelfde.

9.1.2. Compgof.hs

De functie uit de tweede groep zijn :

mapST	(a -> ST s b) -> [a] -> ST s [b]
data Either a b	
data ExitCode	
data IOError13	
either	(a -> c) -> (b -> c) -> (Either a b) -> c
getCPUTime	IO(Int)
listST	[ST a b] -> ST a [b]
primIOToIO	a -> a
returnPrimIO	a -> IO a
system	String -> IO (ExitCode)
thenPrimIO	ST a b -> (b -> ST a c) -> ST a c
type IOGHC a	ST RealWorld (Either IOError13 a)
toInt	a -> a

De datatypen *Either*, *IOError13* en *IOGHC* zijn gedefinieerd om de "luxe" IO van de GHC-compiler te kunnen nabootsen. De functies mapST en listST zijn standaard in de GHC-compiler. De functies *returnPrimIO*, *thenPrimIO* en *either* zijn gedefinieerd om hiermee de functie *readAFile* etc. uit **commutil.hs** mogelijk te maken. Voor meer informatie zie [6]

De functies uit de derde groep zijn :

(>>=)	ST a b -> (b -> ST a c) -> ST a c
(>>)	ST a () -> ST a b -> ST a b
return	a -> ST s a
putStr	String -> IO()
putChar	Char -> IO()
getChar	IO(Char)
newArray	Ix a => (a,a) -> b -> ST c (MutArr c a b)
readArray	Ix b => MutArr a b c -> b -> ST a c
writeArray	Ix b => MutArr a b c -> b -> c -> ST a ()

Van deze functies heeft Gofer een evenknie die alleen een andere naam moet krijgen. Zie hiervoor de handleiding bij Gofer, [8]. Zo is de functie *putChar* gedefinieerd als :

```
putChar :: Char -> IO ()
putChar = putchar
```

En de functie (>>=) is gedefinieerd als :

```
(>>=) :: ST a b -> (b -> ST a c) -> ST a c
(>>=) = thenST
```

De functies uit de vierde groep zijn :

type MR x	= MutVar RealWorld x
type TestPattern	= MA Int Bool
type MA b c	= MutArr RealWorld b c
readAAFile	String -> IOGHC String
writeAAFile	String -> String -> IOGHC String

9.2 SSUTIL.HS

De functies in deze module zijn eenvoudig en hoeven niet verder te worden uitgelegd. In `write(ln)` wordt gebruik gemaakt van de ingebouwde functie `putStr`. De functies `listST_` en `mapST_` zijn uitgebreid besproken in § 2.3.3. De functies `present` en `find` zijn besproken in § 7.3.1.

(>>==)	Typering hangt af van Gofer of GHC
(>>)	Typering hangt af van Gofer of GHC
intstostring	[Int] -> String
stringtoint	String -> Int
inttostring	Int -> String
inttostring'	Int -> String
isnumber	String -> Bool
toUpperString	String -> String
fst4	(a,b,c,d) -> a
snd4	(a,b,c,d) -> b
thd4	(a,b,c,d)-> c
frt4	(a,b,c,d) -> d
writeln	String -> IO ()
write	String -> IO ()
anyTrue	[Bool] -> Bool
anyFalse	[Bool] -> Bool
allTrue	[Bool] -> Bool
allFalse	[Bool] -> Bool
present	(a -> Bool) -> [a] -> Bool
find	(a -> Bool) -> [a] -> a
replace	(a-> Bool) -> (a -> a) -> [a] -> [a]
listST_ sts	zie § 2.3.3
mapST_ f xs	zie § 2.3.3
showExitCode	ExitCode -> IO()
showError	IOError13 -> String

De functie *showError* wordt gebruikt om de foutmelding van een GHC-foutmelding te laten zien. Hiervoor is in de GHC-compiler een data-constructor *IOError13* gedefinieerd met voor iedere mogelijke fout een aparte tak. De structuur van deze functie is als volgt :

```
showError x = case x of
        AlreadyExists msg              -> msg
        HardwareFault msg              -> msg
        IllegalOperation msg           -> msg
        .....                             ....
        UserError msg                  -> msg
        EOF                            -> "End of file"
```

De functie *showExitCode* wordt gebruikt om het resultaat van een functie met het type :

```
..... -> IO (ExitCode)
```

te laten zien. De functie is gedefinieerd als :

```
showExitCode :: ExitCode -> IO()
showExitCode exitcode =
        if isErrorCode exitcode
        then
              writeln ("Error in system : Code = "++
                    (inttostring (getErrorCode exitcode)))
        else
              return ()
 where isErrorCode (ExitSucces)      = False
        isErrorCode (ExitFailure x)  = True
        getErrorCode (ExitFailure x) = x
```

9.3 COMMUTIL.HS

De functies in deze module hebben meer uitleg nodig omdat een aantal gebruik maakt van specifieke mogelijkheden van de GHC-compiler. Omdat ze werken met de voorlopige uitgave van Haskell 1.3 kan hier nog het een en ander veranderen.

userAnswer	IO Bool
readIn	IO String
readAFile	FileName -> IO(Bool,String)
writeAFile	FileName -> FileName -> IO(Bool,String)
existFile	FileName -> IO(Bool)
startEditor	FileName -> Int -> Int -> IO()
getStat	IO()
writeArgs	Arguments -> IO()
getNewName	String -> IO(String,Bool)
whileWithWarning	((a,Bool) -> Bool) -> IO(a,Bool) -> ((a,Bool) -> IO()) -> IO(a,Bool)
for	Int -> (Int -> IO()) -> IO()
forlist	[a] -> (Int -> a -> IO()) -> IO()
rmExistFile	String -> IO()

De functies `userAnswer` en `readIn` zijn al besproken in § 2.3.4. Bij de functies `readAFile`, `writeAFile` en `existFile` en `startEditor` wordt gebruik gemaakt van de twee typen `IO` die de GHC-compiler kent, namelijk `primIO` en `IO`. In § 3.4 zijn deze twee vormen van `IO` ook besproken. Ter herrinering : het type `primIO` is de `IO` van Gofer. Dit betekend dat `primIO` gedefinieerd kan zijn als[36] :

```
type PrimIO a = ST RealWorld a
```

Het type IO is in de voorlopige Haskell 1.3 versie [6] gedefinieerd als :

```
type IO a = PrimIO (Either IOError a).
```

Dit uitgewerkt geeft :

```
IO a = ST RealWorld (Either IOError a)
```

Voor de GHC-compiler zijn de typen nog weer wat anders[37], namelijk :

```
type IO a = PrimIO (Either IOError13 a )
```

De data-constructor `Either` is gedefinieerd als :

```
data Either a b = Left a | Right b
```

De `Left` tak wordt gebruikt voor foutmeldingen. De `Right` tak wordt gebruikt als er geen fout optreedt.

Met deze kennis is het niet moeilijk om de functies te begrijpen. De functie `readAFile` neemt als argument een filename van het type string en levert een aangepaste state en een 2-tuple op. Het eerste element van dit tuple is een boolean die aangeeft of de leesoperatie gelukt is. Het tweede element is een string. Als de operatie gelukt is bevat de string de inhoud van de file. Is de operatie niet gelukt dan bevat de string de foutmelding.

De functie begint met het toepassen van de functie `readFile` via de functie `readAAFile` uit **`compghc.hs`** . Het type van deze functie is :

```
readFile :: String -> IO String
                        { → PrimIO (Either IOError13 String) )
```

Het resultaat is van het type PrimIO. Daarom wordt in `readAFile` de functie `` `thenPrimIO` `` gebruikt. Het type is :

```
thenPrimIO :: PrimIO a -> (a -> PrimIO b) -> PrimIO b
```

Ingevuld voor deze toepassing is dit :

```
PrimIO (Either IOError13 String) ->
  ((Either IOError13 String) -> PrimIO b ) -> PrimIO b
```

Met de functie `either` kan bepaald welke tak van de data-constructor `Either` gebruikt is. De definitie van `either` is :

```
either :: (a-> c) -> (b -> c) -> (Either a b) -> c
either f g (Left x)  = f x
either f g (Right x) = g x
```

---

[36] Of dit de exacte definitie is, is niet bekend.
[37] Zoals eerder gezegd is de Haskell wereld behoorlijk in beweging

Na het toepassen van `readFile` wordt met de functie `either` bepaald welke tak er opgeleverd is door `readFile` en wordt het juiste 2-tuple opgeleverd. Om weer naar het luxe type IO te gaan moet hiervoor de constructie :

```
primIOtoIO ( returnPrimIO ( ... ))
```

gebruikt worden.

Als er een fout optreedt is het resultaat van de functie `readFile` de `Left` tak van `Either`. Dit is van het type `IOError13`[38]. Dit is een data-constructor met veel takken waarvan iedere tak een bepaalde fout voorstelt. De functie `showError` haalt uit deze takken de foutmelding.

Met al deze informatie moeten de functies `readAFile`, `writeAFile`,`existFile`, `rmExistFile` en `startEditor` duidelijk zijn.

De functies `for` en `forlist` zijn twee varianten van een for-lus. Het verschil tussen beiden is de functie die gebruikt wordt.

```
forlist :: [a] -> (Int -> a -> IO()) -> IO()
forlist list f = flist (length list) list f
  where flist _     []  _ = return ()
        flist n (x:xs) f = f n x >> flist (n-1) xs f

for :: Int -> (Int -> IO()) -> IO()
for 0 f =
       return ()
for n f =
       f n >> for (n-1) f
```

De functie `whileWithWarning` is een variant van de while-lus. Deze functie gaat net zolang door totdat niet meer aan de guard voldaan wordt. Wordt aan de guard voldaan dan wordt een melding gegeven.

De functie `getNewName` vraagt een nieuwe naam en gaat net zo lang door tot een nieuwe naam is ingevoerd, of totdat de gebruiker op de escape toets heeft gedrukt. Het resultaat van de functie is een paar met als eerste element een string en als tweede element een boolean. Is op de escape toests gedrukt, dan is de boolean False en is de string leeg. Is niet op de escape toest gedrukt, dan is de boolean True en is de string de nieuwe naam.

De functie `getAllFile` is een functie die werkt met de database van de commandoshell. Met deze functie worden alle definities uit één file uit de database gehaald.

```
{cb}
      getAllFile :: MR DesignData -> FileName -> IO[Elem Name]
      getAllFile designDB filename =
        getSysDefListFile   designDB filename >>= \syslist   ->
        getTupleDefListFile designDB filename >>= \tuplelist ->
        getBaseDefListFile  designDB filename >>= \baselist  ->
        getAtomDefListFile  designDB filename >>= \atomlist  ->
        return ((map getBase baselist) ++
                (map getAtom atomlist) ++
                (map getTuple tuplelist) ++
                (map getSystem syslist))

{ce}
```

De functie `savefile` wordt gebruikt om een file te bewaren. Er moeten twee filenamen als argument meegegeven worden waarvan de tweede eventueel leeg kan zijn. Is het tweede argument een lege string dan wordt de file met dezelfde naam bewaard. Anders wordt de file onder de andere naam bewaard.

---

[38] Dit is weer specifiek GHC

```
{cb}
      savefile :: MR DesignData -> FileName -> FileName -> IO()
      savefile designDB fnsource fntarget =
       getAllFile designDB fnsource >>= \result ->
       (case fntarget of
        ""        -> writeAFile fnsource (pp1 result)
        otherwise -> writeAFile fntarget (pp1 result)) >>= \ioTuple ->
       case (fst ioTuple) of
       True  -> case fntarget of
                ""        -> writeln ( "File " ++
                                       fnsource ++
                                       "saved")
                otherwise -> writeln ( "File " ++
                                       fnsource ++
                                       " saved as " ++
                                       fntarget)
       False -> writeln (snd ioTuple) >>
                writeln "File not saved"

{ce}
```

De functie *remove_file* verwijderd een file uit de database.

```
{cb}
      remove_file :: MR DesignData -> FileName -> IO()
      remove_file designDB filename =
        remInfoItem         designDB filename >>
        remSysDefListFile   designDB filename >>
        remTupleDefListFile designDB filename >>
        remBaseDefListFile  designDB filename >>
        remAtomDefListFile  designDB filename

{ce}
```

Met de functie *rmExistFile* wordt een file gewist als die file bestaat. Met de functie *system* uit de GHC-compiler wordt de file verwijderd.

10 HOOFDPROGRAMMA COMMANDOSHELL

Een belangrijke functie in de commandoshell is de herhaling waarin de invoer van de gebruiker ingelezen wordt en het gekozen commando wordt uitgevoerd. Dit wordt het hoofdprogramma genoemd, alhoewel in de functionele wereld programma's niet voorkomen. Wel expressies, zodat hoofdexpressie misschien een betere naam is. Het doel van dit deel van de commandoshell is in ieder geval wel duidelijk. De samenstellende files van het hoofdprogramma zijn :

- **commain.hs**
- **commrdln.hs**
- **comm.hs**

10.1 DATA CONSTRUCTOREN VOOR HET HOOFDPROGRAMMA

In de module **commmain.hs** wordt een data-constructor *Command* gedefinieerd. Deze constructor heeft voor ieder commando een tak. Verder worden enkele constanten gedefinieerd. Hier is verder geen uitleg nodig.

```
defaultLenHistory = 20

defaultprompt = "\n$"

type Costpair    = (String)
type ECostpair   = (String,Int)
type EECostpair  = (String,Int,Int)
type EEECostpair = (String,Int,Int,String)

type Arguments  = [String]

data Command =
               | Behave
               | Cost
               | FDesign
               .....
               .....
               | ShowState
               | Skipcom
               | Statistics
               | Xilinx
```

10.2 INLEZEN VAN COMMANDO

In de module **commrdln.hs** worden functies gedefinieerd die de invoer van de gebruiker inlezen en analyseren. De functie *readcommandline* heeft als argument de database van de commandoshell en levert een 3-tuple op. Het eerste element van dit tuple is de codering van het ingevoerde commando met een element van de data-constructor *Command* uit § 10.1. Het tweede element is de string weergave van het commando, zeg maar het eerste woord dat de gebruiker ingevoerd heeft. Het derde argument zijn de argumenten die ingevoerd zijn.

Bij een invoer *"ss -f abs1 abs2"* wordt dit 3-tuple :

```
(ShowState, "ss", ["-f", "abs1","abs2"])
```

De functie *readcommandline* leest de regel in met de functie *readln*. In de variabele *comline* staat het resultaat van deze functie. Als de lengte van *comline* 0 is doordat direct op enter gedrukt is, wordt het 3-tuple :

```
(Nocomm,"",[])
```

Is de lengte van de string langer dan 0 dan wordt met de functie *isEnvItemLoaded* bepaalt of in de environment van het programma een "alias" voorkomt van hetgeen ingevoerd is. Hiermee kunnen lange opdrachten een handige naam gegeven worden. Is dit zo dan wordt de ingevoerde string vervangen door de associatie waarde van deze alias. Is er geen alias dan blijft de ingevoerde string gelijk. De (vervangen) string wordt aangeboden aan de functie *process_command*.

Met de functie *startArglex* wordt de invoer gescand en verdeeld in tokens. Met de functie *extractCommand* wordt bepaald welke tak van de data-constructor *Command* gekozen moet worden. Deze functie is niet moeilijk. Daarna wordt de lijst van tokens gesplitst in de *head* en de *tail*. De *head* is de string representatie van het commando, de *tail* is de lijst met argumenten.

```
readcommandline :: MR DesignData -> IO (Command, Commandd, Arguments)
readcommandline designDB =
      readln >>=  \comline ->
       case (length comline) of
       0          -> return (Nocomm,"",[])
       otherwise -> isEnvItemLoaded designDB comline >>= \value ->
                    if value
                    then
                        getEnvItem designDB comline >>= \envvalue ->
                        process_command designDB (getEnvValue envvalue)
                    else
                        process_command designDB comline

process_command :: MR DesignData -> String -> IO

      (Command,Commandd,Arguments)
process_command designDB comline =
      addCommandItem designDB comline >>
      return (extractCommand (head (startArglex comline)),
              head (startArglex comline),
              tail (startArglex comline)
             )

extractCommand :: String -> Command
extractCommand s = case (toUpperString s) of
                    "!"        -> ShellCom
                    "?"        -> Help
                    "B"        -> Save
                    "C"        -> Cost
                    "FD"       -> FDesign
                    .....        -> ....
                    .....        -> ....
                    "R"          -> RepeatL
                    "RM"         -> Remove
                    "S"          -> Setpar
                    "SS"         -> ShowState
                    "ST"         -> Statistics
                    "X"          -> Xilinx
                    otherwise    -> Nocomm
```

10.3 HOOFDPROGRAMMA

De functie *main*, uit de file **comm.hs**, is de functie waarmee begonnen wordt te evalueren als het gecompileerde programma wordt opgestart. Dit is een verplichting van de GHC-compiler, er moet een functie **main** zijn. De functie begint met een aankondiging en maakt daarna een nieuwe variabele in de state. In deze variabele staat de database. De functie *mkInitDataBase* staat beschreven in § 7.3.1. Daarna wordt overgegaan naar de ontwerp-lus in de functie *designloop*. Deze functie doet achter elkaar het volgende :

- zet een prompt met de functie *setprompt*. De prompt die gezet wordt, kan worden ingesteld door in het environment een associatie paar (*prompt,promptstring*) in te voeren. Is zo'n paar niet aanwezig in de environment dan wordt de defaultprompt genomen, zie § 10.1
- leest de commando regel.
- bepaalt het commando en de argumenten.
- voert de actie uit met de functie *select_action*.

Dit is alles, net alsof je imperatief aan het programmeren bent.

```
main :: IO ()
main = writeln "Begin sessie \n "   >>
       mkInitDataBase               >>= \initdatabase ->
       newVar initdatabase          >>==  \initdb ->
       designloop initdb

setprompt :: MR DesignData -> IO()
setprompt designDB =
    isEnvItemLoaded designDB "prompt" >>= \value ->
    if value
    then
        getEnvItem designDB "prompt"       >>= \envpair ->
        write (getEnvValue envpair)
    else
        write (defaultprompt)

designloop :: MR DesignData -> IO()
designloop designDB =
       setprompt designDB           >>
       readcommandline designDB     >>= \c    ->
       seperateCommand c            >>= \com  ->
       seperateCommand_string c     >>= \coms ->
       seperateArgs c               >>= \args ->
       selectAction designDB com coms args
```

De functie *selectAction* voert het gewenste commando uit. Wordt als enige argument de string *"-?"* meegegeven aan het commando dan wordt de help van het commando getoond. Anders wordt het commando uitgevoerd. Deze functie is een hele mooie demonstratie van State Transformers met sequentiering en state. Na het uitvoeren van het commando wordt de functie *designloop* weer aangeroepen. Het commando *"quit"* roept *designloop* niet meer aan waardoor de herhaling stopt.

```
selectAction :: MR DesignData -> Command -> Commandd ->
                                          Arguments -> IO()
selectAction designDB com coms args =
          if (args /= []) && (head args == "-?")
          then

     case com of
          Behave      -> behavehelp                    >> designloop designDB
          Cost        -> costhelp                      >> designloop designDB
          FDesign     -> file_designhelp               >> designloop designDB
          ......      -> ......
          Xilinx      -> xilinxshelp                   >> designloop designDB
          Nocomm      ->                  designloop designDB
          Skipcom     ->                  designloop designDB

          else

          case com of
          Behave      -> behavecom designDB args    >> designloop designDB
          Cost        -> costcom designDB args      >> designloop designDB
          FDesign     -> file_designcom designDB args
                                                    >> designloop designDB
          .......     -> .......
          Xilinx      -> xilinxscom designDB args   >> designloop designDB
          RepeatL     ->                  repeatcom designDB args
          Skipcom     ->                  designloop designDB
          Quit        ->                  quitcom designDB args
```

De functie *repeatcom* herhaalt een eerder ingevoerd commando. De functie werkt als volgt :

- Als geen argumenten worden ingevoerd dan wordt het laatste commando herhaalt. Met de database-functie *getLastCommand* wordt de ingevoerde string uit de database gehaald.
- Worden wel argumenten ingevoerd dan moeten er twee argumenten worden ingevoerd, nl. de string *!!* en een getal. Het getal geeft aan welk commando uit de history-lijst herhaald moet worden.
    - Is dit zo dan wordt met de database functie *getCommandIndex* de betreffende entry uit de database gehaald.
    - Is het tweede argument geen getal dan wordt het laatste[39] commando uitgevoerd.
    - Is het eerste argument niet !! dan wordt een foutmelding gegeven
    - Worden teveel of te weinig argumenten ingevoerd dan wordt een foutmelding gegeven
- Als geldige invoer gegeven wordt dan levert dit de gewenste string op die ingevoerd is in het verleden. Deze string wordt aangeboden aan de functie *repeat_action*.

De functie *repeat_action* lijkt sterk op *readCommandLine* uit § 10.2. De werking is duidelijk. Eén opmerking is wel op zijn plaats. Als de gebruiker een commando uit het verleden uitzoekt dat ook een herhaal-commando is, kan op twee manieren gereageerd worden :

- is het een herhaal-commando zonder argumenten, dan wordt dit uitgelegd als een *skip*-commando. Dit om te voorkomen dat het programma vast loopt in een oneindige lus.
- is het een herhaal-commando met argumenten, dan wordt *repeatcom* weer opnieuw aangeroepen, nu met de argumenten uit het verleden. Hiermee is een indirectie in het herhaal-commando mogelijk.

[39] Dat wil zeggen, het laatste commando voor het repeat commando.

```
repeatcom :: MR DesignData -> Arguments -> IO ()
repeatcom designDB args
 | l_arg == 0 =
      getLastCommand designDB >>= \comline ->
       if comline /= ""
       then
            repeat_action designDB comline
            else
             selectAction designDB Skipcom "" args
 | l_arg == 2 && f_arg == "!!" && isnumber s_arg =
      getCommandIndex designDB (stringtoint s_arg)>>= \comline ->
       if comline /= ""
            then
                 repeat_action designDB comline
            else
                 selectAction designDB Skipcom "" args
 | l_arg == 2 && firstarg == "!!" =
      writeln "Geen getal ingevoerd als index"
 | otherwise = writeln "Verkeerde argumenten" >> designloop designDB
  where l_arg = length args
        f_arg = head args
        s_arg = head (tail args)

repeat_action :: MR DesignData -> Commandd -> IO()
repeat_action designDB comline = pre_select_action designDB com args
  where com  = extractCommand (head (startArglex comline))
        coms = head (startArglex comline)
        args = tail (startArglex comline)
        pre_select_action designDB com args =
            case com of
            RepeatL   -> if length args == 0
                         then
                              selectAction designDB Skipcom coms args
                         else
                              repeatcom designDB args
            otherwise -> selectAction designDB com coms args
```

11 DE COMMANDO'S

In dit hoofdstuk worden de implementaties besproken van commando's uit de commando set. De implementaties staan in de files :

- **commdesg.hs** : file voor File Design commando
- **commedit.hs** : file voor File Edit commando
- **commhist.hs** : file voor History commando
- **commload.hs** : file voor Load commando
- **commlsys.hs** : file voor Show State commando
- **commprad.hs** : file voor Print Internal commando
- **commpret.hs** : file voor Print Text commando
- **commquit.hs** : file voor Quit commando
- **commremv.hs** : file voor Remove commando
- **commrest.hs** : file met rest commando's
- **commsave.hs** : file voor Save commando
- **commset.hs** : file voor Set commando
- **commshel.hs** : file voor Shell escape commando
- **commtrns.hs** : file voor Transformation commando
- **commxili.hs** : file voor Xilinx commando

Globaal kan de structuur van een commando beschreven worden als :

- een hoofdfunctie die als eerste wordt aangeroepen. Dit is de functie die in `selectAction` uit § 10.3 wordt aangeroepen[40].
- een aantal ondersteunende functies

In de hoofdfunctie wordt aan de hand van de argumenten geanalyseerd welke variant van het commando de gebruiker wil. Een voorbeeld is figuur 36 :

```
showstate designDB arg
  | length arg == 0 = mapST_ ss [ "-F","-S","-T","-B","-A" ]
  | otherwise       = mapST_ ss arg
 where ss onearg
        | onearg == "-F" = showLoadedFiles   designDB
        | onearg == "-S" = showLoadedSystems designDB
        | onearg == "-T" = showLoadedTuples  designDB
        | onearg == "-B" = showLoadedBases   designDB
        | onearg == "-A" = showLoadedAtoms   designDB
```

Figuur 36: Voorbeeld van analyse van commando variant.

Dit voorbeeld is genomen uit het commando `ShowState` dat (delen van) de inhoud van de database laat zien. Als in dit voorbeeld geen argumenten gegeven worden betekent dit dat de gebruiker alles wil zien. In dit geval wordt met de State Transformer `mapST_` en een lijst van fictieve argumenten de lokale functie `ss` aangeroepen voor ieder element uit de fictieve argumenten lijst. In de functie `ss` wordt met een verzameling guarded expressies gekeken welke tak uitgevoerd moet worden. In iedere tak worden weer hulp functies aangeroepen. In dit geval `showLoadedFiles`, `showLoadedSystems` enzovoort.

Bij sommige functies is maar één variant mogelijk. Dan nog kunnen meerdere argumenten worden opgegeven. De argumenten worden dan ieder afzonderlijk gezien als één argument bij één aanroep van het commando. Een voorbeeld is het laad commando. Dit kent één variant, het laden van een file, en één argument, de naam van de te laden file. Toch kunnen meerdere filenamen worden opgegeven. Iedere filenaam hoort dan bij één laad commando. Bijvoorbeeld :

[40] Net alsof het imperatief programmeren is.

```
l file1 file2 file3 .. file-n.
```

De invoer wordt gezien als :

```
l file1 ";" l file2 ";" l file3 ";" ..... ";" l file-n
```

De analyse van de argumenten is als volgt :

```
loadcom :: MR DesignData -> Arguments -> IO ()
loadcom designDB arg
    | length arg == 0 = writeln "geen filenaam ingevoerd"
    | otherwise       = mapST_ (loadpart designDB) arg
```

De hulpfuncties van de commando's zijn minder eenduidig opgebouwd. Er zijn in ieder geval hulpfuncties om de verschillende varianten van het commando uit te voeren. Verder is er niet veel overeenkomstig te vinden. In de volgende paragraaf wordt het file design commando uitgelegd.

11.1 FILE DESIGN COMMANDO

Dit commando is voor het ontwerpen van een systeem door het systeem te beschrijven "als een programma tekst" met een editor. Het commando start de editor op waarin het systeem ingevoerd kan worden. Wordt de editor beëindigd dan wordt geprobeerd de file in te lezen. Bij dit commando is het mogelijk meerdere file namen als argument op te geven. Voor iedere filenaam afzonderlijk wordt het commando herhaalt. Deze herhaling wordt bereikt met de State Transformer *mapST_* en de functie *design_file*.

In de functie *design_file* wordt bepaald of de ingevoerde filenaam al in de database aanwezig is of dat er al een file bestaat met dezelfde naam. Is dit niet aanwezig dan wordt de editor gestart met de filenaam als argument. Na het beëindigen van de editor wordt gevraagd of de file geladen moet worden. Moet de file geladen worden dan wordt met de functie *loadcom* uit de module **comnload.hs** geprobeerd de file te laden.

Is de file wel in de database aanwezig of bestaat er al een file met deze naam dan wordt het commando niet uitgevoerd. De filenaam moet de naam zijn van een nieuw bestand.

11.2 FILE EDIT COMMANDO

Dit commando komt sterk overeen met het file-design commando. Een verschil is echter dat het nu de file aanwezig moet zijn in de database. Eerst wordt de file uit de database verwijderd met de functies *getAllFile* en *remove_file*. Hiermee wordt voorkomen dat de database inconsistent wordt. Het resultaat van *getAllFile* wordt met de pretty printer voor programma tekst naar een tijdelijke file geschreven. Deze tijdelijke file wordt geladen in de editor. Bij het beëindigen wordt de file geladen met het commando *loadcom* uit de file **comnload.hs**.

11.3 HISTORY COMMANDO

Het history commando is eenvoudig. Uitleg is overbodig.

11.4 LOAD COMMANDO

De functie *loadcom* is de hoofdfunctie van deze module. Voor iedere string in de rij argumenten wordt de hulp functie *loadpart* uitgevoerd.

De functie *loadpart* probeert een file te laden. Eerst wordt in de database gekeken of de file al geladen is. Als dit zo is wordt een fout melding gegeven en wordt de functie beëindigd. Als dit niet zo is wordt geprobeerd of de file geparst kan worden met de functie *fp* uit de module **ssparser.hs**. Het resultaat van deze functie is van het type *ParserValue*. Dit is een data-constructor met 3 takken (zie ook § 6.2.2) :

⇒ Is het resultaat de `Error` tak dan is de foutmelding al gegeven in de functie `fp`. Daarom is in dat geval alleen `return ()` nodig.

⇒ Is het resultaat de `NotValid` tak dan wordt gevraagd of de file geladen moet worden in een editor zodat direct aan de file gewerkt kan worden. Kiest de gebruiker hiervoor dan wordt de editor gestart. In het geval van emacs als editor wordt gesprongen naar de regel waar de fout is opgetreden. Voor andere editoren die eveneens die mogelijkheid bieden moet dit nog geprogrammeerd worden.

⇒ Is het resultaat de `Valid` tak dan voldoet de inhoud van de file aan de grammatica van de Systeem Subset en kan de interne weergave worden verwerkt in de database met de database functies. De geladen definities worden met delen van de pretty printer uit § 6.2.2 op het scherm getoond.

Voor het verwerken van de definities in de database wordt de volgende constructie gebruikt :

```
addSysDefList   designDB (systlist newfile)
systlist xs =
            map (\y -> (filename,getADSSysName y,False,y))
                (filter isSysDef   (eR xs))
```

Deze constructie is voor de systeem definities. Voor de andere soorten definities zijn soortgelijke functies gemaakt. De variabele `xs` in deze functie is het resultaat van de functie `fp`. In dit geval is dit resultaat van de vorm `Valid prog`. Met de functie

```
(eR xs)
```

wordt `prog` uit deze data-constructor gehaald. De functie

```
(filter isSysDef (eR xs))
```

filtert de systeem definities uit de interne weergave. De functie `isSysDef` staat in de file **`ssad.hs`**, zie § 5.1. In de "argument"-functie van de `map` functie wordt een systeem definitie gemaakt zoals die in de database gebruikt wordt, zie § 7.1 punt 3. Met `getADSSysName` uit § 5.1 wordt de identifier-string uit de interne data structuur gehaald.

De nieuwe systeem definities worden op het scherm gezet met de volgende functie :

```
pp2IO (slist newfile)
slist xs = filter isSysDef (eR xs)
```

De functie `pp2IO` is de print variant die niet sorteert en de string direct op het scherm zet, zie § 6.2.2. De functie `eR` is dezelfde als eerder besproken. De functie `slist xs` filtert de systeem definities uit de interne weergave.

11.5 Print Internal Commando

Met het commando "pa" wordt een voor mensen te begrijpen weergave gemaakt van de interne weergave van een definitie. De "echte[41]" interne opslag uit figuur 37 wordt met het commando zoals getoond in figuur 38.

[41] Voor zover mogelijk. Het gaat om het idee.

```
[Sys_def(Id(FunctionName"s")NoType,NTuple[Ident(Id(ArgName"f")
NoType),Ident(Id(ArgName"g")NoType),Ident(Id(ArgName"x")NoType
)],EAp(EAp(EName(Id(FunctionName"f")NoType))(EName(Id(Function
Name"x")NoType)))(ATuple[EAp(EName(Id(FunctionName"g")NoType))
(EName(Id(FunctionName"x")NoType))]),[]),Tuple_def(Id(TupleWit
hName"a1")NoType,Ident(Id(TupleName"a1")NoType),EAp(EAp(EName(
Id(FunctionName"and")NoType))(EName(Id(FunctionName"true")NoTy
pe)))(EName(Id(FunctionName"true")NoType)),[])]
```

Figuur 37 : De "echte" interne weergave van twee definities.

omgezet in :

```
[ Sys_def ( Id (FunctionName "s") NoType,
            NTuple [Ident (Id (ArgName "f") NoType),
                    Ident (Id (ArgName "g") NoType),
                    Ident (Id (ArgName "x") NoType)
                   ],
            EAp (EAp (EName (Id (FunctionName "f") NoType))
                     (EName (Id (FunctionName "x") NoType)))
                (ATuple [EAp (EName (Id (FunctionName "g") NoType))
                             (EName (Id (FunctionName "x") NoType))
                        ]),
            []
          ),
  Tuple_def( Id (TupleWithName "a1") NoType,
             Ident (Id (TupleName "a1") NoType),
             EAp (EAp (EName (Id (FunctionName "and") NoType))
                      (EName (Id (FunctionName "true") NoType)))
                 (EName (Id (FunctionName "true") NoType)),
             []
           )
]
```

Figuur 38 : Het resultaat van het commando "pa" toegepast op de structuur uit figuur 37.

Deze module maakt gebruik van de pretty printer uit § 6.2.3. Het is een front-end voor de functies uit de module **ssppad.hs**.

De mogelijke varianten van dit commando zijn :

- pa -@ ⇒ alle definities in de database in de volgorde :
  1. basis type-definties.
  2. atomaire type-definties.
  3. systeem definities.
  4. tuple definities.
- pa -C ⇒ alle definities van het huidige systeem.
- pa -C -O ⇒ alle definities van het huidige systeem gesorteerd. De volgorde is gelijk aan de volgorde bij de variant pa -@.
- pa -F f-naam ⇒ alle definities van de file met de naam f-naam.
- pa -F -O f-naam ⇒ alle definities van de file met de naam f-naam gesorteerd. De volgorde is. gelijk aan de volgorde bij de variant pa -@.
- pa -S s-naam ⇒ alle systeem definities met de naam s-naam.
- pa -T t-naam ⇒ alle tuple definities met de naam t-naam.

- pa -B b-naam ⇒ alle basis type-definties met de b-naam
- pa -A a-naam ⇒ alle atomaire type-definties met de naam a-naam
- pa -F -S f-naam s-naam ⇒ één systeem definitie met naam s-naam uit file f-naam
- pa -F -T f-naam t-naam ⇒ één tuple definitie met naam t-naam uit file f-naam
- pa -F -B f-naam b-naam ⇒ één basis type-defintie met naam b-naam uit file f-naam
- pa -F -A f-naam a-naam ⇒ één atomaire type-defintie met naam a-naam uit file f-naam

In de hoofdfunctie wordt aan de hand van de argumenten de juiste hulp functie gekozen.

Voor iedere variant is een functie gemaakt. Deze functies lijken moeilijker dan ze zijn. Het is vaak een heel gepuzzel om het resultaat netjes op het scherm te krijgen. Met de database functies worden de gewenste definities uit de database gehaald waarna ze geprint worden met de pretty print functies. Wanneer de uitvoer gesorteerd moet worden, wordt de functie `adpp1IO` gebruikt. Moet de uitvoer niet gesorteerd worden dan wordt de functie `adpp2IO` gebruikt.

11.6 PRINT TEXT COMMANDO

Met het commando "p" wordt een tekstuele weergave gemaakt van de interne weergave van een definitie. Zo wordt de interne structuur uit figuur 37 uit § 11.5 omgezet in :

```
s (f,g,x) = f x (g x);
a1 = and true true
```

Dit hoeft niet precies gelijk te zijn aan hetgeen in de file staat. Deze module maakt gebruik van de pretty printer uit § 6.2.2. Het is een front-end voor de functies uit de module **`sspp.hs`**. De werking is grotendeels gelijk aan de werking van vorige module uit § 11.5. Alleen de manier waarop de interne weergave wordt gepresenteerd verschilt. Daarom wordt deze module verder niet besproken.

11.7 SHOWSTATE COMMANDO

Met het `Show State` commando wordt de "inhoudsopgave" van de database getoond. De definities worden niet getoond, wel de namen van de geladen definities. Verder wordt, indien van toepassing, getoond of een onderdeel veranderd is sinds het inlezen ervan. Zo kan een systeem definitie veranderd worden doordat er een transformatie regel op toegepast is. En een file kan zijn veranderd doordat een van de systeem definities uit de file veranderd is.

De mogelijke varianten zijn :
- `ss` ⇒ Laat de hele inhoudsopgave zien
- `ss -F` ⇒ Laat de file zien
- `ss -S` ⇒ Laat de systeem definities zien
- `ss -T` ⇒ Laat de tuple definities zien
- `ss -B` ⇒ Laat de basis type-definties zien
- `ss -A` ⇒ Laat de atomaire definities zien

Het is mogelijk combinaties van deze varianten te maken, bijvoorbeeld :

```
ss -A -B -F.
```

Een dergelijk commando wordt uitgevoerd door ieder argument als een argument aan de functie aan te bieden. In dit geval :

```
ss -A -B -F ⇔ ss -A ";" ss -B ";" ss -F
```

met de ";" van Pascal als notatie geleend.

Wordt geen argument ingevoerd dan wordt in de hoofdfunctie een lijst van argumenten toegevoegd. Welke van deze varianten uitgevoerd moet worden wordt in een lokale definitie van de hoofdfunctie uitgezocht.

De opbouw van de hulpfuncties is als volgt :

- Met een database functie wordt de verzameling gevraagde definities opgehaald.
- Het resultaat hiervan wordt aangeboden, met een State Transformer, aan een hulp functie die er een lijst van *Iseq's* van maakt. De hulpfunctie maakt gebruik van de *Iseq* functies beschreven in § 6.2.1. Deze lijst wordt met de functie *iDisplayIO* op het scherm gezet.

Een voorbeeld hiervan is :

```
showLoadedSystems designDB =
        getSysDefList designDB >>= \syslist ->
        iDisplayIO (ppsys iStr syslist )
```

Met de database functie *getSysDefList* wordt de verzameling systeem definities opgehaald. Van deze lijst wordt een lijst van *Iseq's* gemaakt in de functie *ppsys (pretty print systeem)*. Het resultaat hiervan wordt met *iDisplayIO* op het scherm gezet. De rest van de functies zijn ook zo opgebouwd.

11.8 Quit Commando

Met het commando quit wordt het programma beëindigd. Het commando quit kent twee varianten. De ene variant bewaart zonder te vragen alle veranderde systemen in files en handhaaft hierbij de naam. De andere variant kijkt of er systemen veranderd zijn. Is dit zo dan wordt voor ieder veranderd systeem gevraagd of dit bewaard moet worden.

11.9 Remove Commando

Met het Remove commando kunnen ingelezen files worden verwijderd. Het is hierbij belangrijk dat het "systeem", database en filesysteem, consistent blijven. Worden aan de hoofdfunctie geen argumenten meegegeven dan wordt een foutmelding gegeven. Worden meerdere argumenten meegegeven dan wordt voor ieder van deze argumenten hetzelfde gedaan. Vergelijk § 11.7.

In de functie *reminfo* wordt een file uit de database verwijderd. Eerst wordt gekeken of de ingevoerde filenaam wel in de database aanwezig is. Is dit niet zo dan wordt een fout melding gegeven. Is dit wel zo dan wordt gekeken of elementen van de file veranderd zijn. Is dit niet zo, dan wordt de file met al zijn definities uit de database verwijderd. Is dit wel zo dan wordt gevraagd of de veranderde file bewaard moet worden.

11.10 Rest Commando's

In deze module staan de commando's die al wel in de commando set zijn opgenomen maar nog niet zijn uitgewerkt. Om een goed overzicht te houden zijn deze functies in één file gezet.

11.11 Save Commando

Het Save commando wordt gebruikt om een file te bewaren. Hierbij is het belangrijk om het "systeem", database en file systeem, consistent te houden. In de hoofdfunctie wordt aan de hand van de argumenten bepaald welke variant van het commando uitgevoerd moet worden.

- Worden geen argumenten meegegeven dan wordt een foutmelding gegeven.
- Wordt één argument meegegeven dan is de werking als volgt :
  - Eerst wordt met een database functie gekeken of de file wel ingelezen is.

- Is dit niet zo dan wordt een foutmelding gegeven en stopt de functie.
- Is dit wel zo dan wordt gekeken of er al een file is die dezelfde naam heeft en of de file veranderd is.
  - Is beiden het geval dan wordt gevraagd of de bestaande file overschreven moet worden met de nieuwe inhoud. De reden hiervoor is dat niet zomaar een file overschreven kan worden.
    - Is het antwoord ja dan wordt de file weggeschreven met de functie `savefile`.
    - Is het antwoord nee dan wordt een nieuwe naam gevraagd, met de functie `getNewName`, waarna de file wordt weggeschreven onder deze nieuwe naam. Als deze nieuwe naam hetzelfde is als de oude naam wordt daar niet tegen gewaarschuwd. Of als er een al een file bestaat met deze nieuwe naam wordt daar ook niet tegen gewaarschuwd.
  - Is een van beide controles niet waar dan kan zonder gevaar de file bewaard worden onder de aangegeven naam.
- Worden twee argumenten meegegeven dan wordt de ingelezen file onder een andere naam bewaard. Er wordt nog niet gecontroleerd of er een file is met de nieuwe naam.
- Meer dan twee argumenten geeft een foutmelding.

11.12 SET COMMANDO

Met het Set commando wordt de lokale environment beheerd. Deze functie is simpel. De enige opmerking die gemaakt wordt heeft betrekking op het veranderen van de environment. Wordt één argument gegeven dan wordt het associatiepaar uit de environment verwijderd, als dit voorkomt. Worden twee argumenten gegeven dan wordt gekeken of er al een associatie in de environment aanwezig. Is dit niet zo dan wordt het paar toegevoegd. Is dit wel zo dan wordt gevraagd of de oude waarde overschreven moet worden.

11.13 SHELL COMMANDO

Dit commando wordt gebruikt om een shell opdracht uit te voeren. Dit gaat niet altijd goed. En `cd` opdracht wordt bijvoorbeeld niet uitgevoerd. Waarom is niet duidelijk. Het kan zijn dat het programma afdwingt dat in dezelfde directory gebleven wordt tijdens het runnen van het programma. Maar een `ls` opdracht werkt wel goed. Zelfs `startx` (in LINUX) werkt. Het commando is meer gemaakt om te laten zien wat er allemaal kan met de functie `system` dan als echt nuttige toepassing. Maar wie weet wat de toekomst gaat brengen.

Wel opvallend is het gebruik van `iDisplay` in dit geval. Hiermee wordt de rij argumenten weer als één string samengevoegd met een spatie tussen ieder argument.

11.14 TEMPLATE VOOR HET TRANSFORMATIE COMMANDO

Dit is nog niet een compleet commando. De code in deze file dient als een template voor hoe een transformatie commando geïmplementeerd zou kunnen worden.

11.15 XILINX COMMANDO

Met de functie `xilinxscom` wordt de variant gekozen die de gebruiker wil. Hiervoor worden de functies `xilinxCurrentFile`, `xilinxFileSys` en `xilinxFileTuple` gebruikt.

12 DE BETEKENIS FUNCTIE "XILINX"

Met deze betekenisfunctie kan een ontwerp worden omgezet in formaat waarmee het getest kan worden met het Xilinx pakket. De eerste implementatie van deze betekenisfunctie is gemaakt als een "standalone" versie in Gofer in een laboratorium 9 project. Dit project is uitgevoerd door J.T.W. Vischer en H. Drenth. Deze implementatie is bewerkt zodat het resultaat past in de ontwikkelomgeving.

In dit hoofdstuk wordt de bewerkte implementatie besproken. Het verslag van het lab. 9 project dient als basis voor de bespreking en alleen de belangrijke verschillen worden besproken. De eerste versie wordt in de rest van dit hoofdstuk aangeduid met versie_1. De geïncorporeerde versie wordt versie_2 genoemd.

12.1 FILE VERSCHILLEN

In versie_1 werd gewerkt met onderstaande files :

- **absyn.hs** ⇒ Datastructuur
- **lexer.hs** ⇒ Scanner
- **parse.hs** ⇒ Parserfuncties
- **corepars.hs** ⇒ Parser CoreProgram
- **eval.hs** ⇒ Evaluator van expressies
- **lam.hs** ⇒ Lambda lifter
- **norm.hs** ⇒ Normalisator
- **gen.hs** ⇒ XNF-uitvoer generator
- **fileio.hs** ⇒ I/O functies
- **voorbeel.hs** ⇒ Testfile

In versie_2 blijven alleen de onderstaande files over. Voor de andere files, zoals bijvoorbeeld de parser, wordt gebruik gemaakt van de commandoshell.

- **xilidata.hs** ⇒ Datastructuur voor foutafhandeling.
- **xilieval.hs** ⇒ Evaluator van expressies.
- **xililam.hs** ⇒ Lambda lifter.
- **xilinorm.hs** ⇒ Normalisator.
- **xiligen.hs** ⇒ XNF-uitvoer generator.
- **commxili.hs** ⇒ "Hoofdprogramma". Dit is beschreven in § 11.15.

12.2 SYNTAX VERSCHILLEN

De taal waarmee de systemen beschreven worden verschillen van elkaar. In versie_1 wordt gewerkt met de grammatica uit figuur 39. In versie_2 wordt gewerkt met de grammatica uit § 4.2.3. Het blijkt echter dat, voor gebruik in de Xilinx functie, een groot deel van de beide grammatica's op een kleine conversie na overeenkomt.

```
Prog         ::= Def [ ";" Prog ]
Def          ::= FunctionDef | TupleDef
FunctionDef  ::= FunctionName Tuple "=" Expr
TupleDef     ::= Tuple "=" Expr
FunctionName ::= identifier
Tuple        ::= VarName | "(" Tuple [ "," Tuple ] ")"
VarName      ::= identifier
Expr         ::= ( Tuple | Application ) [ "where" WhereDef ]
Application  ::= FunctionName Atuple
Atuple       ::= Tuple | Application | "(" Atuple [ "," Atuple ] ")"
WhereDef     ::= Def [ ";" WhereDef ]
```

Figuur 39 : De grammatica uit versie_1.

12.3 DATA STRUCTUUR VERSCHILLEN

Belangrijker dan de verschillen in de grammatica's zijn de verschillen in de datastructuren waarin een ontwerp wordt opgeslagen. Hiermee wordt immers gewerkt. In tabel 36 staan de belangrijkste verschillen.

(1)	
`type Program = [ScDefn]`	`type Tekst a = [Elem a]`

(2)	
`type ScDefn = (FieTup,VarTuple,Expr)`	`data Elem a = Base a |Atom a |Sys_def (a,NTuple a,Expr a,Tekst a) |Tuple_def (a,NTuple a,Expr a,Tekst a)`

(3)	
`data Tuple a = TVar a | TTup [Tuple a] | TNil`	`data NTuple a = Ident a | NTuple [NTuple a]`

(4)	
`data Expr = EVar Name | EAp FieName Expr | ETuple ExprTuple | EWhere Expr [ ScDefn ]`	`data Expr a = EName a | EAp (Expr a) (Expr a) | ELam (NTuple a) (Expr a) | ATuple [Expr a]`

(5)	
`type Name = String`	`data Name = Id IdDetails IdType`

Tabel 36 : De verschillen in de data structuur met overeenkomende variant.

1. Dit verschil en de invloed hiervan op de code is beperkt. Door de juiste tekstverwerker acties is versie_1 op dit punt om te zetten in versie_2.

2. Dit verschil is te verkleinen omdat de constructor alternatieven `"Base a"` en `"Atom a"` uit versie_2 in de Xilinx functie geen betekenis hebben en dus ook weggelaten kunnen worden in de code van versie_2.

   I. In versie_1 wordt het onderscheid tussen een systeemdefinitie en een tupledefinitie gemaakt aan de hand van een expressie van het type `VarTuple`. Is deze expressie `TNil`, waarmee gecodeerd wordt dat er geen argumenten zijn, dan is het een tuple definitie, anders een systeem definitie. In versie_2 wordt gewerkt met twee aparte data-constructoren. Werd in de code van versie_1 gewerkt met TNil op het tweede element, dan is in versie_2 gewerkt met de data-constructor Tuple_def. Analoog voor het andere geval.

   II. Het eerste element in versie_1, van het type `FieTup`, wordt gebruikt voor de naamgeving van een "systeem[42]"-definitie of voor het opslaan van het tuple bij een tupledefinitie. In versie_2 wordt de naam van een definitie opgeslagen in het eerste element van het type *a*. De argumenten, in het geval van een systeem definitie, worden opgeslagen in het tweede element van het type NTuple a. Het tuple, in het geval van een tuple definitie, wordt ook in dit tweede element opgeslagen. Het onderscheid, nogmaals, wordt gemaakt aan de hand van de data-constructoren.

   III. Het derde element `Expr` is in versie_1 zo opgebouwd dat hierin ook de where-clausulus opgeslagen kunnen worden. In versie_2 zijn deze twee onderdelen van een expressie uit elkaar gehaald en worden

[42] In versie_2 terminologie een systeem definitie. In versie_1 wordt dit een functie definitie genoemd

ze in respectievelijk het derde en vierde element opgeslagen. Dit onderscheid wordt later nog verder besproken bij punt 4.

3. De eerste twee takken van de beide data-constructoren zijn bijna rechtstreeks in elkaar om te zetten. De laatste taak van versie_1 wordt in versie_2 niet gebruikt en kan daarom uit de code van versie_1 gehaald worden.

4. In dit deel zitten een aantal grote verschillen.
   I. De where-clausules worden in versie_1 in een expressie opgeslagen. In versie_2 wordt de expressie en de where-clausules apart opgeslagen. Dit geeft de volgende transformatie :

```
(naam,tuple,(EWhere e1 where_expr))   ⇒
                    Sys_def     (naam,tuple,expr,where_expr) of
                    Tuple_def   (naam,tuple,expr,where_expr)
```

   Bij de functie *eval_exp* uit versie_1 is de match-tak, *EWhere e1 e2*, verwijderd en er is een extra parameter in de functie *eval_exp* gebracht waarin de where-clausules doorgegeven kunnen worden zodat die gebruikt kunnen worden in de overige takken. De where-clausules worden nu niet meer in *eval_exp* "voor het eerst ontdekt" maar in de functie *eval_sc*.

   II. In versie_1 is de applicatie tak, *EAp FieName Expr*, concreter gekozen. In versie_2 is meer mogelijk. Daarom moet in versie_2 rekening gehouden worden met deze vrijheid. Een transformatie is als volgt :

```
EAp FieName Expr  ⇒ EAp (EName a) (Expr a)
```

   Dit is een rechtstreekse transformatie van versie_1 in versie_2 en is in versie_2 dit geval aanwezig in een expressie dan moet de bijbehorende code uit versie_1 worden uitgevoerd. In versie_2 is bijvoorbeeld ook mogelijk :

```
EAp (EAp e1 e2) e3.
```

   Nu moet andere code worden uitgevoerd en daarom wordt in versie_2 onderscheidt gemaakt tussen deze twee gevallen in *eval_exp*.

   III. In versie_2 zijn ook lambda expressies mogelijk. In versie_1 kan dit niet. Tot nu toe worden lambda expressie in versie_2 NIET ondersteund. Dit moet nog gemaakt worden.

5. Dit verschil is beperkt. In de typering in versie_2 moet overal rekening gehouden worden met *Name*.

12.4 FOUT AFHANDELING

In versie_1 is geen rekening gehouden met de mogelijkheid dat een onderdeel niet geladen is. Met de functie *get* uit **xilieval.hs** wordt de lijst van definities doorzocht maar als een definitie niet voorkomt dan wordt een zinloos iets opgeleverd. In versie_2 is voor het netjes afhandelen van fouten de data-constructor *XV* gedefinieerd :

```
data XV a b =  XGood a
             | XFail b

type XValue a = XV a String
```

Deze data-constructor komt overeen met de data-constructor *ParserValue* uit § 6.1 en de constructor *Maybe* uit de GHC-bibliotheek. Met de typevariabele *a* kan het type *XValue* getypeerd worden, bijvoorbeeld *XValue (Elem Name)*. In een groot deel van de code wordt gewerkt met dit type.

Als voorbeeld de functie *xilinxFile* uit **xiligen.hs**. Dit is de main-functie waarmee het xilinx-proces gestart wordt. In deze functie wordt de functie *enorm_sc* aangeroepen. Als het resultaat van deze functie *XGood value* is, dan wordt de uitvoer gemaakt met de functie *makeOutPutString*. Is het resultaat *XFail msg*, dan wordt de string *msg* afgedrukt.

12.5 DE DATABASE EN OPHALEN UIT DE DATABASE

In versie_1 is voor het opslaan van alle definities is geen aparte voorziening gemaakt. De "database" is ingebakken in de code[43]. In versie_2 is het Xilinx programma een onderdeel van de commandoshell waardoor in versie_2 de database apart komt. Dit vereist op dit punt belangrijke aanpassingen aan de code van versie_1.

De database uit versie_2 is beschreven in hoofdstuk 7 en is van het type : *MR DesignData*. De onderdelen die van belang zijn voor het Xilinx programma zijn : SysDef (§ 7.3 en § 7.4.4) en TupleDef (§ 7.3 en § 7.4.5).

In de functie *comxilinx* uit **commxili.hs** wordt met de juiste database-functies de lijst van systeem- en tuple- definities uit de database gehaald en aangeboden aan de functie *xilinxFile* uit **xiligen.hs**[44] Deze lijst van definities is van het type :

```
[(FileName, PartName, PartChanged, Elem Name)]
```

Dit lijkt niet op de benodigde typering in versie_1 waar (*FieTup, Tuple Name, Expr*) vervangen moet worden door alleen *Elem Name*. Hiermee is, waar nodig, rekening gehouden. Bijvoorbeeld in de functie *rename* wordt daarom uit de lijst van onderdelen een lijst gemaakt met alleen de vierde elementen. Deze lijst wordt bewerkt en later weer samengevoegd.

Met de functie *get* uit **xilieval.hs** kan een definitie uit de lijst met onderdelen gehaald worden. De regels voor het zoeken zijn als volgt :

- Eerst wordt gekeken of de definitie in de where-clausules zit. Is dit zo dan wordt opgeleverd :
  XGood (definitie)
- Is dit niet zo, dan wordt gekeken of de definitie in dezelfde file zit als de definitie waar op dit moment mee gewerkt wordt. Is dit zo dan wordt opgeleverd :
  XGood (definitie)
- Is dit niet zo, dan wordt de eerste definitie uit de database gehaald die de gewenste naam heeft. Is er zo'n definitie dan wordt weer opgeleverd :
  XGood (definitie)
- Is er niet zo'n definitie dan wordt *XFail* met de juiste boodschap opgeleverd en stop de bewerking.

12.6 DE CODE

Tot zover de verschillen. Met deze extra uitleg en het verslag van versie_1 moet het mogelijk zijn om de code te begrijpen. Als laatste kan worden opgemerkt dat sommige code uit versie_1 verplaats is naar andere files in het project omdat ze daar logisch hoorden. Een voorbeeld zijn de functies *convert* en *deconvert*. Deze functies zijn verplaatst naar de file **ssad.hs**. Tevens zijn een aantal functies vervallen. Of omdat ze niet meer nodig waren, of omdat ze sowieso niet nodig waren. Voorbeelden hiervan zijn de functie *tuple_concat* en *binder(sOf)*. Van de code worden alleen die functies besproken die nieuw zijn in versie_2.

---

[43] Dit is de lijst : *[(FieTup, Tuple Name, Expr)]*, bijvoorbeeld uit de functie *eval_exp*.

[44] Er is niet gekozen voor het meegeven van de hele database, *MR DesignData*, omdat hiermee een groot deel van de code in "State Transformer" vorm geschreven zou moeten worden.

12.6.1 Xilidata.hs

Met de functies *getXValue* en *getXFail* kan het resultaat worden verkregen van een berekening. De lijst *xnf_list* is de lijst van standaard ingebouwde elementen die het Xilinx pakket kent. Een definitie uit deze lijst krijgt een speciale behandeling. Met de functie *xWrong* kan bepaalt worden of een bewerking gelukt is of niet.

```
module XilinxData where

data XV a b = XGood a
            | XFail b

type XValue a = XV a String

xnf_list = ["OR","NOT","AND","NOR","NAND"]

getXGood :: XValue a -> a
getXGood (XGood value) = value

getXFail :: XValue a -> String
getXFail (XFail msg) = msg

xWrong :: XValue a -> Bool
xWrong (XGood x)   = False
xWrong (XFail msg) = True
```

12.6.2 Xilieval.hs

In deze module de functies :

get	FileName -> PartName -> [Elem Name] -> [PartDef] -> XValue (Elem Name)
eval_exp	Expr Name -> NTuple Name -> [Elem Name] -> [PartDef] -> FileName -> XValue (Expr Name)
eval_tuple	[Expr Name] -> NTuple Name -> [Elem Name] -> [PartDef] -> FileName -> XValue [Expr Name]
replaceVar	(Expr Name,NTuple Name,Expr Name) -> Expr Name
eval_sc	Elem Name -> [PartDef] -> FileName -> XValue (Elem Name)

Deze functies zijn allemaal beschreven in [10]

12.6.3 Xiligen.hs

inputs	NTuple Name -> [(String,String)]
outputs	Expr Name -> [(String,String)]
signals	Elem Name -> [(String,String)]
inputpins	Expr Name -> [(String,String)]

find_xnfs	[Elem Name] -> Int -> [(String,[(String,String)])]
symbloks	Elem Name -> [(String,[(String,String)])]
mk_strpindefs	[(String,String)] -> Int -> String
mk_strsymbloks	[(String,[(String,String)])] -> Int -> String
mk_strsigs	[(String,a)] -> Int -> String
outputsymbloks	Elem Name -> String
outputsignals	Elem Name -> String
outputstring	Elem Name -> String
makeOutPutString	MR DesignData -> String -> Elem Name -> IO ()
xilinxFile	MR DesignData -> String -> String -> [PartDef] -> IO()

Van deze lijst zijn de functies `makeOutPutString` en `xilinxFile` nieuw. Met `makeOutPutString` wordt het eigenlijke genereren van xilinx-uitvoer gestart. Er wordt een file geschreven met de juiste naam en indien het associatie paar `(verbose,on)` in de environment zit wordt het resultaat ook op het scherm gezet. De functie `xilinxFile` regelt het bepalen van de naam van de xilinx file en werkt de foutafhandeling af.

12.6.4 Xililam.hs

getLamNames	Int -> [String] -> (Int,[String])
makeName	String -> Int -> String
mapAccuml	(a -> e -> b -> (a,c)) -> a -> e -> [b] -> (a,[c])
rename	[PartDef] -> [PartDef]
rename_sc	Int -> [(String,String)] -> Elem Name -> (Int,Elem Name)
newNames	Int -> [String] -> (Int,[String],[(String,String)])
lookup	[(String,String)] -> String -> String
rename_e	[(String,String)] -> Int ->Expr Name -> (Int,Expr Name)
rename_w	[(String,String)] -> Int -> Expr Name -> [Elem Name] -> (Int,Expr Name,[Elem Name])
rename_list	[(String,String)] -> Int -> [Expr Name] -> (Int, [Expr Name])
ell	FileName -> PartName -> [PartDef] -> XValue (Elem Name)

De functies `mapAccuml`, `rename` en `rename_w` zijn aangepast respectievelijk nieuw. De functie `getLamNames` is dezelfde als de functie `getNames` in versie_1. `mapAccuml` is aangepast omdat de gebruikte functie, het eerste argument, andere argumenten heeft. De functie `rename` is aangepast aan het nieuwe type `PartDef`. De functie `rename_w` rekent af met het verschil uit § 12.3, punt 4. Door op een hoger niveau de where-clausules te verwerken is dit verschil uit de wereld geholpen.

12.6.5 Xilinorm.hs

normalize	Int -> Expr Name -> (Expr Name,[Elem Name],Int)
norm_tuple	[Expr Name] -> Int -> (Expr Name,[Elem Name],Int)
cb	(Expr Name,[a],b) -> (Expr Name ,[a],b) -> (Expr Name,[a],b)
norm_e	Elem Name -> Elem Name
norm_sc	FileName -> PartName -> [PartDef] -> XValue (Elem Name)
enorm_sc	FileName -> PartName -> [PartDef] -> XValue (Elem Name)

De *cb* is een andere naam voor de functie *combine* uit versie_1.

13 DE BETEKENIS FUNCTIE "KOSTEN"

De code voor de betekenisfunctie kosten beslaat twee files :

- `commcost.hs`
- `costmain.hs`

De kostenfunctie is op dit moment zeer simpel. Eigenlijk is de implementatie ervan een beetje blijven liggen. Het enige dat nu gedaan wordt, is het opsommen van de elementen in de definitie. Met argumenten en / of where-clausules wordt niets gedaan. Sorry, maar helaas.

13.1 DEFINITIE VAN DE KOSTEN FUNCTIE

De kosten functie $\mathcal{K}$ is, op basis van de abstracte datastructuur, gedefinieerd als :

$$\mathcal{K} \in \text{"ADS van deSysteem Subset"} \rightarrow \text{Kosten}.$$

De representatie van het bereik *Kosten* is een rij van functienamen. Dan wordt $\mathcal{K}$:

$$\mathcal{K} \in \text{"ADS van de Systeem Subset"} \rightarrow [String]$$

$\mathcal{K}$ [[Tekst a]] = $\mathcal{K}$ [[[elem_list]]] = *Sum*($\mathcal{K}$ [[elem_list]])

$\mathcal{K}$ [[Elem a]] = $\mathcal{K}$ [[Base a]] = []

$\mathcal{K}$ [[Atom a]] = []

$\mathcal{K}$ [[Sys_def (a, NTuple a, Expr a, Tekst a)]] = $\mathcal{K}$[[Expr a]] + $\mathcal{K}$ [[Tekst a]]

$\mathcal{K}$ [[Tuple_def (a, NTuple a, Expr a, Tekst a)]] = $\mathcal{K}$[[Expr a]] + $\mathcal{K}$[[Tekst a]]

$\mathcal{K}$ [[EName name]] = $\mathcal{K}$ [[name]]

$\mathcal{K}$ [[EAp e1 e2]] = $\mathcal{K}$ [[e1]] + $\mathcal{K}$[[e2]]

$\mathcal{K}$ [[ELam tuple e1]] = $\mathcal{K}$ [[e1]]

$\mathcal{K}$ [[ATuple [e_list]]] = $\mathcal{K}$ [[[e_list]]] = *Sum*($\mathcal{K}$ [[e_list]])

$\mathcal{K}$ [[Name]] = $\mathcal{K}$ [[IdDetails]]

$\mathcal{K}$ [[IdDetails]] = $\mathcal{K}$ [[TupleWithNoName]] = []

$\mathcal{K}$ [[TupleWithName _]] = []

$\mathcal{K}$ [[BaseName _]] = []

$\mathcal{K}$ [[AtomName _]] = []

$\mathcal{K}$ [[FunctionName n]] = [n]

$\mathcal{K}$ [[ArgName _]] = []

$\mathcal{K}$ [[TupleName _]] = []

In deze definitie is *Sum* te definieren als :

```
map (++) k_list
```

13.2 DE MODULE COMMCOST.HS

De functies in deze module zijn :

costcom	MR DesignData -> Arguments -> IO()
ppCostOfCurrent	MR DesignData -> PartName -> IO()
ppCostOfFile	MR DesignData -> FileName -> IO ()
addCommonCost	MR DesignData -> ECostpair -> IO ()
ppCommonCost	MR DesignData -> IO()
saveCommonCost	MR DesignData -> IO()

De functie `costcom` is de functie die uitzoekt welke variant de gebruiker wil. De functie `ppCostOfCurrent` geeft de kosten van een onderdeel uit de huidige file. De functie `ppCostOfFile` geeft de kosten van alle systeem en tuple definities uit een file. Met `addCommonCost` kan voor een onderdeel de defaultkosten worden ingevoerd in de database. Is er al een waarde voor het onderdeel dan wordt gevraag of de oude waarde overschreven moet worden met de nieuwe waarde. `ppCommonCost` drukt de verzamling defaultkosten uit de database af. En als laatste bewaart de functie `saveCommonCost` de verzamling defaultkosten uit de database in de file **`commonct.lst`**.

13.3 DE MODULE COSTMAIN.HS

costOfOnePart	MR DesignData -> FileName -> PartName -> IO()
showStandardCost	MR DesignData -> PartName -> IO()
writeCostTekst	MR DesignData -> (FileName,PartName,Bool,Elem Name) -> [String] -> IO ()
standardcost	[ECostpair] -> [DefaultCost] -> IO([EECostpair])
compress	[Costpair] -> [ECostpair] -> [ECostpair]
spelem	Costpair -> [ECostpair] -> Bool
addone	Costpair -> [ECostpair] -> [ECostpair]
costTekst	[Elem Name] -> [Costpair]
costElem	Elem Name -> [Costpair]
costSys_def	(Name, NTuple Name, Expr Name, [Elem Name]) -> [Costpair]
costTuple_def	(Name, NTuple Name, Expr Name, [Elem Name]) -> [Costpair]
costExpr	Expr Name -> [Costpair]
costName	Name -> [Costpair]

- De functie `costOfOnePart` berekent, verrassing, dekosten van een onderdeel. Met de functies `isPartItemLoadedFileName` wordt bepaalt of een onderdeel met de juiste namen geladen is. Is dit niet zo dan wordt een melding gegeven. Is dit wel zo dan wordt de lijst van samenstellende onderdelen getoond.
- De functie `showStandardCost` laat de defaultkosten van een standaard onderdeel zien.
- `writeCostTekst` is de functie die `costOfOnePart` aanroept om het resultaat te laten zien.
- `standaardCost` krijgt als argument

- een lijst met paren met daarin de gevonden functienamen en het aantal keren dat een naam voorkomt in de definitie.
- de lijst met defaultkosten.

Aan de hand hiervan rekent `standaardCost` uit hoeveel de standaardonderdelen in de definitie kosten. Het resultaaat is een lijst met 3-tuples waarvan het eerste element de naam is, het tweede element het aantal voorkomens is in de definities en het derde element de kosten bevat.

- `compress`, `spelem` en `addone` comprimeren een lijst van functienamen tot een lijst van paren waarin iedere functienaam maar een keer voorkomt en waar in het tweede element het aantal voorkomens in de definitie staat. Bijvoorbeeld :

```
["and","not","not","and","xor"]
```

is de lijst van functienamen die in de definitie voorkomt. Dan is het resultaat van `compress` :

```
[("and",2),("not",2),("xor",1)]
```

- `costTekst`, `costElem`, `costSys_def`, `costTuple_def`, `costExpr` en `costName` zijn rechtstreekse implementaties van de kosten functie zoals die gedefinieerd is in § 13.1

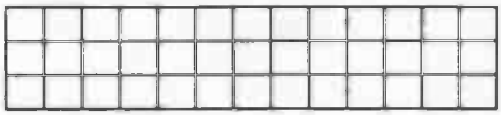
14 TEST PATROON GENERATOR

Voor het testen van het gedrag[45] van een ontwerp kan gebruik worden gemaakt van timingsdiagrammen. Er is een programma gemaakt waarmee timingsdiagrammen gemaakt kunnen worden. Deze diagrammen worden naar een functionele beschrijving omgezet en kunnen hierna gebruikt worden voor een gedragsfunctie.

14.1 INLEIDING

Het programma is grafisch opgezet. Na het kiezen van het "Pattern"-commando in de commandoshell krijgt de gebruiker een nieuw window. Op dit window zijn een aantal onderdelen te onderscheiden :

- een menubalk bovenin
- een statusbalk onderin
- een "invoergebied". In dit gebied kan een timingsdiagram worden gemaakt. Het invoergebied is onderverdeeld in rechthoeken zoals figuur 40 laat zien.



Figuur 40 : Het invoergebied.

De rijen in dit gebied stellen de "invoerpinnen" voor van het ontwerp. De kolommen de opeenvolgende tijdstippen. Door te klikken op vlak *(i,j)* wordt voor invoerpin i op tijdstip j de waarde hoog of laag gezet.

- een "uitvoergebied". In dit gebied kan het resultaat getoond worden van een simulatie. Dit wordt in de huidige opzet nog niet ondersteunt. Dit gebied ziet er net zo uit als het invoergebied.
- Ghost gebied. Dit is het gebied dat niet behoort tot een van de andere gebieden. Als hierin geklikt wordt gebeurt er niks.

Bij het kiezen van het commando "Pattern" heeft de gebruiker een aantal opties. Deze zijn :

- `PAT -N naam x y z`. Hiermee wordt een "neutraal" timingsdiagram gemaakt. Neutraal omdat dit niet bij een ontwerp hoort.
  - De naam die wordt opgegeven is de naam waaronder de functionele beschrijving wordt opgeslagen. De file krijgt de toevoeging `".pat"`.
  - De x, y en z zijn optioneel en zijn alle drie een positief geheel getal. Als y wordt opgegeven dan moet x ook worden opgegeven. Analoog voor z. Met x kan het aantal invoerpinnen gekozen worden, met y het aantal tijdstappen en met z de grootte van de vlakken.
- `PAT -S snaam y z`. Hiermee wordt voor een systeem met de naam `snaam` uit de huidige default file een window geopend. Hierbij wordt geprobeerd het invoer- en uitvoergebied te bepalen aan de hand van de definitie. Dit werkt als volgt :
  - Is de typering gegeven dan kan volledig het aantal invoer- en uitvoerpinnen bepaalt worden. Bij de invoerpinnen komt de naam te staan van de argumenten. Bij de uitvoerpinnen de naam van de typering.
  - Is de typering niet gegeven dan kan alleen het aantal invoerpinnen bepaalt worden aan de hand van het aantal argumenten. Er wordt nu geen uitvoer gebied geopend.

  De argumenten y en z zijn weer optioneel en zijn dezelfde als boven.
  De functionele beschrijving wordt opgeslagen onder de naam :

  ```
  "naam van default file"++"."++"naam van systeem"++."pat"
  ```
- `PAT -T tnaam y z`. Hetzelfde als boven, alleen nu voor een tuple met de naam `tnaam` uit de huidige default file.

---

[45] Dit is een semantische functie zoals die voorkomt in de System Semantics uit § 1.4. In termen van de System Semantics is het type van deze functie m : $m \in \mathcal{S} \rightarrow \mathcal{D}$ waarin $\mathcal{D}$ het betekenis domein $\mathcal{Gedrag}$ is en $\mathcal{S}$ een verzameling systeembeschrijvingen.

- *PAT -F fnaam -S snaam y z*. Hetzelfde als boven, alleen nu voor een systeem met de naam *snaam* uit de file *fnaam*.
- *PAT -F fnaam -T tnaam y z*. Hetzelfde als boven, alleen nu voor een tuple met de naam *tnaam* uit de file *fnaam*.

Bij het aanroepen van het "Pattern"-commando wordt gekeken of er al een "pattern-file" is met de juiste naam. Is deze file er al, dan wordt die ingelezen en verwerkt zodat begonnen wordt met een eerder opgeslagen patroon. Is de inhoud van de file niet overeenkomstig de bijbehorende grammatica, dan wordt een foutmelding gegeven en kan de editor geopend worden. Wordt de editor geopend dan wordt na het sluiten van de editor opnieuw geprobeerd om de inhoud in te lezen. Dit wordt herhaald net zolang tot het inlezen lukt of totdat de gebruiker ervoor kiest om door te gaan. In dit laatste geval wordt begonnen met een "leeg" invoergebied, dit wil zeggen dat alle waarden laag zijn.

Bij het afsluiten van het commando wordt gewaarschuwd indien een patroon veranderd is sinds de laatste keer dat het patroon is opgeslagen. Hierdoor kan niet per ongeluk een patroon worden verloren.

De menu balk kent de volgende opties :

- redraw ⇒ Met deze optie wordt het scherm opnieuw opgebouwd.
- reset ⇒ Hiermee worden alle invoerwaarden op laag gezet en wordt het scherm opnieuw opgebouwd.
- make output ⇒ Hiermee wordt een functionele beschrijving gemaakt van het testpatroon.
- blok / wave ⇒ Hiermee kan gekozen worden uit twee verschillende representaties van het testpatroon.
- copy ⇒ Hiermee kan een deel van het patroon worden gekopieerd. Het te kopiëren deel moet op een regel liggen en kan daarna op ieder vlak in het invoergebied worden geplakt. Is de lengte van het te kopiëren deel te lang dan wordt dit afgekapt.
- quit ⇒ Hiermee wordt het patroonprogramma beëindigt. Is het patroon sinds de laatste keer van opslaan nog veranderd, dan wordt hiervan een melding gegeven.

In de nu volgende paragraven wordt de code besproken.

14.2 De Code

Bij het commando behoren de volgende files :

- **patdata.hs** ⇒ In deze file staan de gebruikte datatypen. Dit zijn voornamelijk datatypen voor het werken met in grote variabele window's.
- **patfunc.hs** ⇒ In deze file staat de functie waarmee de functionele beschrijving van een patroon gemaakt wordt.
- **patstate.hs** ⇒ In deze file staan de functies en datatypen waarin het patroon intern wordt opgeslagen.
- **patpars.hs** ⇒ In deze file staat de parser voor de patronen files. Dit is het omgekeerde van de functies uit **patfunc.hs**.
- **patwutil.hs** ⇒ Een verzameling grafische routines (vandaar de w van windows).
- **patputil.hs** ⇒ Een verzameling patroon routines (vandaar de p van patroon).
- **commpatt.hs** ⇒ In deze file staan de hoofdfuncties.

Naast deze files wordt ook gebruik gemaakt van de files **win.c**, **win.h** en **Win.hs**. Deze files zijn eerder gemaakt door Rinze Krol in een laboratorium 9 project [4]. Aan de file **Win.hs** zijn de volgende functies toegevoegd :

- `ringBell percentage = primIOToIO (_ccall_ ring_bell percentage)`
- `getcurrentcolor = primIOToIO (_ccall_ get_current_color)`
- `getcurrentlinestyle = primIOToIO (_ccall_ get_current_line_style)`
- `setlinestyle c = primIOToIO (_ccall_ set_line_style c)`

Wat ze doen, spreekt voor zich. Aan de file **win.c** zijn de bijbehorende functies *ring_bell*, *get_current_color*, *get_current_line_style* en *set_line_style* toegevoegd.

14.2.1 PatData.hs

Deze module bevat de constanten en de datatypen, en de daarbij behorende access-functies, die gebruikt worden bij het "Pattern"-commando. Er zijn datatypen gedefinieerd voor :

- *Het omvattende window.*

  In een groot aantal functies uit het "Pattern"-commando, is het nodig om de grootte van het window te kennen. Als de grootte constant is kunnen hiervoor vaste constante expressies genomen worden[46]. Maar de grootte van het window is in het "Pattern"-commando variabel en daarom moeten de huidige waarden meegegeven worden aan deze functies.

  Dit is bereikt door een extra parameter[47] te gebruiken in deze functies. Dit is een parameter van het type `Window`. In een expressie van dit type staan de gegevens die nodig zijn om met het window te kunnen werken.

  De data-constructor `Window` bevat de volgende elementen :

  - `Width` ⇒ breedte van het window
  - `Height` ⇒ hoogte van het window
  - `XSides`[48] ⇒ het aantal tijdstappen
  - `YSides` ⇒ het totale aantal rijen in het invoer- en uitvoergebied
  - `Num_I` ⇒ het aantal rijen in het invoergebied
  - `Num_O` ⇒ het aantal rijen in het uitvoergebied
  - `TStep` ⇒ het aantal tijdstappen
  - `TSize` ⇒ de grootte van een vlak

  Dit datatype wordt verpakt in een State Transformer zodat de waarden van de elementen kunnen veranderen. In het bijzonder kunnen ze bij het begin van een executie van het "Pattern"-commando geinitialiseerd worden. Dit is precies wat gewenst is.

  Bij de access-functies is gebruik gemaakt van de typering om te kunnen laten zien wat er gebeurd in de functies. Een `get...` functie haalt informatie uit de expressie, een `set...` functie verandert informatie. Door de typering is de semantiek van deze functies zo duidelijk.

- *De invoer en uitvoer gebieden.*

  Doordat deze gebieden variabel zijn in grootte, gelden hiervoor dezelfde beweringen als bij het omvattende window. De gegevens van deze gebieden zijn opgeslagen in expressies van het type `VWindow`. Deze gebieden worden virtuele windows genoemd, vandaar de naam `VWindow`.

  De elementen van deze constructor zijn :

  - `Left_X` ⇒ De x-waarde links boven van een virtueel window in een omvattende window
  - `Left_Y` ⇒ De y-waarde links boven van een virtueel window in een omvattende window
  - `Righ_X` ⇒ De x-waarde rechts onder van een virtueel window in een omvattende window
  - `Righ_Y` ⇒ De y-waarde rechts onder van een virtueel window in een omvattende window
  - `TSize` ⇒ De grootte van een vlak uit een invoer- of uitvoergebied
  - `UnitLen` ⇒ De grootte van een vlak gedeeld door 2

De constanten die gedefinieerd zijn, zijn :

- constanten van het omvattende window.

---

[46] Zoals ook gebeurd in [4]

[47] In een functionele taal zijn geen globale variabelen mogelijk. Dit is een van de voordelen van functioneel programmeren maar in dit geval ook een van de nadelen. Nu is veel extra parameter-tik-werk enzovoort nodig. Met de daarbijhorende fouten. Je geeft steeds een niet veranderd stuk informatie mee dat heel goed in een globale structuur zou kunnen passen.

[48] Dit element is overbodig.

- constanten voor de verschillende kleuren en lijntypen.
- constanten voor default aantal invoer- en uitvoerpinnen, default aantal tijdstappen en default grootte van een vlak in het invoer- en uitvoergebied.
- constanten voor de karakter grootte.
- twee commandolijsten en het aantal commando's in de commandolijsten. Er zijn twee commandolijsten om de wisseloptie "Block / Wave" weer te kunnen geven.

Om tot een beperking van de omvang van het verslag te komen zal alleen de naam en de typering van de functies genoemd worden. De bijbehorende bodies zijn gemakkelijk en het heeft geen nut om ze mee te nemen in het verslag.

getWW	MR Window -> IO Width
getWH	MR Window -> IO Height
setWW	MR Window -> Width -> IO ()
setWH	MR Window -> Height -> IO ()
getWS	MR Window -> IO (Width,Height)
getWX	MR Window -> IO XSides
getWY	MR Window -> IO YSides
setWX	MR Window -> XSides -> IO ()
setWY	MR Window -> YSides -> IO ()
getWI	MR Window -> IO Num_I
getWO	MR Window -> IO Num_O
setWI	MR Window -> Num_I -> IO ()
setWO	MR Window -> Num_O -> IO ()
getWIO	MR Window -> IO (Num_I,Num_O)
getWTST	MR Window -> IO (TStep)
getWTSZ	MR Window -> IO (TSize)
setWTST	MR Window -> TStep -> IO ()
setWTSZ	MR Window -> TSize -> IO ()
getWT	MR Window -> IO (TStep,TSize)

Tabel 37: Functies voor het datatype *Window*.

getLx	MR VWindow -> IO (Left_X)
getLy	MR VWindow -> IO (Left_Y)
getRx	MR VWindow -> IO (Righ_X)
getRy	MR VWindow -> IO (Righ_Y)
getV_SZ	MR VWindow -> IO (TSize)
getV_UN	MR VWindow -> IO (UnitLen)
getVW	MR VWindow -> IO (Left_X,Left_Y,Righ_X,Righ_Y)
getUp	MR VWindow -> IO (Left_X,Left_Y)
setLx	MR VWindow -> Left_X -> IO ()
setLy	MR VWindow -> Left_Y -> IO ()
setRx	MR VWindow -> Righ_X -> IO ()
setRy	MR VWindow -> Righ_Y -> IO ()
setVW	MR VWindow -> (Left_X,Left_Y,Righ_X,Righ_Y,TSize,UnitLen) -> IO()

Tabel 38: Functies voor het datatype *VWindow*.

14.2.2 Patfunc.hs

In deze module de functie *mkfunction*. Deze functie maakt van een testpatroon een "functionele" beschrijving. Een andere manier om tegen deze functie aan te kijken is de functie te beschouwen als een "pretty printer" van het testpatroon. Een voorbeeld hiervan is te zien in figuur 41. Dit is een functionele beschrijving van het patroon :

```
[ [1,0,1,0,1],[0,1,0,1,0],[0,1,1,1,0]]
```

waarin 1 (of True) staat voor "hoog" en 0 (of False) staat voor "laag".

```
casefun1 x =
 case ( mod x 5 ) of
    0 ->  True
    1 ->  False
    2 ->  True
    3 ->  False
    4 ->  True

casefun2 x =
 case ( mod x 5 ) of
    0 ->  False
    1 ->  True
    2 ->  False
    3 ->  True
    4 ->  False

casefun3 x =
 case ( mod x 5 ) of
    0 ->  False
    1 ->  True
    2 ->  True
    3 ->  True
    4 ->  False
```

Figuur 41 : Een voorbeeld van de uitvoer van *mkfunction*.

In figuur 42 is de grammatica van de uitvoer te zien.

```
Start        ::= Funs "Bottom"

Funs         ::= Fun {Fun}      {+ p(name)  +}
                                {- "Bottom" -}

Fun          ::= FunName ArgName "=" Case_guard Case_expr

FunName      ::= identifier

ArgName      ::= identifier

Case_guard   ::= "case"  "("  "mod" ArgName len_number ")" "of"

Case_expr    ::= Case_elem { Case_elem }
                             {+ p(number)        +}
                             {- p(name),"Bottom" -}

Case_elem    ::= number "->" Value

Value        ::= "True"
               | "False"
```

Figuur 42 : De grammatica van de uitvoer. De `Start` produktie is toegevoegd voor het parsen. Voor de BNF-notatie geldt dezelfde legenda als bij de grammatica uit § 4.2.

In de module wordt gebruikt gemaakt van de functies uit de file **ssiseq.hs** zoals beschreven in § 6.2. Van de code wordt alleen de hoofdfunctie *mkfunction* besproken. De rest spreekt voor zich. De functie mkfunction begint met het ophalen van informatie over de windows. Uit deze informatie wordt ondermeer bepaalt hoeveel tijdstappen en invoerrijen er zijn. Dit is belangrijk bij het verkrijgen van de lijsten met booleans. Met de functie *getLines* uit **patstate.hs** wordt uit de State Transformer *tp* een lijst met lijsten van booleans opgeleverd die het testpatroon representeert. Deze wordt aangeboden aan de functie *mkcase_s* die van iedere lijst een case-expressie maakt.

```
mkfunction :: MR Window -> FileName -> MR Bool -> TestPattern -> IO ()
mkfunction win fname changed tp =
        getWI win                       >>= \num_i ->
        getWTST win                     >>= \num_t ->
        getLines num_i num_t tp         >>= \lines ->
        -- nu is lines een lijst van lijsten met booleans
        rmExistFile fname               >>
        writeAFile fname
              (iDisplay (iConcat (mkcase_s iStr 1 lines)))
                                        >>= \iotuple ->
        case iotuple of
        (True,"")    -> changed >>:: False >>
                        writeln ("wrote : "++fname)
        (False,msg) -> writeln msg
```

14.2.3 Patpars.hs

De functie *fpPattern* uit deze module ontleed een functionele beschrijving van een testpatroon en levert de lijst van lijsten met booleans op. Dit is het omgekeerde van de functie *mkfunction* uit **Patfunc.hs**. De gebruikte technieken en functies zijn beschreven in § 6.1. Samen met de grammatica uit figuur 42, § 14.2.2, moet hiermee de code duidelijk zijn.

```
fpPattern :: String -> IO (ParserValue [[Bool]])
fpPattern fname =
        readAFile fname >>= \ioTuple ->
        case (fst ioTuple) of
        True  -> case (patternParser ((snd ioTuple)++" Bottom")) of
                 (One value toks) -> return (Valid value)
                 (Fail msg  toks) -> echo_error msg toks >>
                                     return (NotValid line colomn)
                                      where line   = snd3 (head toks)
                                            colomn = thd3 (head toks)
        False -> writeln (snd ioTuple) >>
                 return (Error)
```

14.2.4 Patputil.hs

In deze module een verzameling utilities, speciaal voor het "Pattern"-commando.

- *black_square :: MR VWindow -> Int -> Int -> IO().*
  Deze functie wordt gebruikt wanneer voor de "block" representatie van het testpatroon is gekozen.
  Met deze functie wordt een vlak in het invoer- of uitvoergebied gekleurd ten teken dat de waarde op dat tijdstip en op die invoerpin "hoog" is. Er wordt een van de acht kleuren gekozen al naar gelang het nummer van de rij. De kleur wit wordt niet gebruikt omdat dit de kleur is die "laag" weergeeft.. In plaats daarvan wordt zwart gekozen. Het vlak wordt gekleurd door het juiste aantal verticale lijnen te tekenen. Hiervoor wordt de functies *for* en *drawVertLine* gebruikt.

BELANGRIJK : Bij het begin van de functie wordt de huidige kleur opgeslagen zodat bij het verlaten van de functie die kleur weer kan worden ingesteld. Dit is in deze module een algemeen gebruikte techniek om te voorkomen dat kleuren opeens gaan veranderen.

- `white_square :: MR VWindow -> Int -> Int -> IO()`
  Deze functie wordt gebruikt wanneer voor de "block" representatie van het testpatroon is gekozen.
  De tegenovergestelde functie van black_square.

- `grid_index :: MR VWindow -> Int -> Int -> IO (Int,Int)`
  Deze functie bepaalt welk vlak in het invoergebied is aangeklikt. Wordt een vlak buiten het invoergebied aangeklikt dan wordt het paar (0,0) opgeleverd. Anders het rij- en kolom- nummer van het vlak

- `in_square :: MR VWindow -> Int -> Int -> IO Bool`
  Deze functie bepaalt of in het invoergebied is aangeklikt.

- `eraseWave :: MR VWindow -> Int -> Int -> IO()`
  Deze functie wordt gebruikt wanneer voor de "wave" representatie van het testpatroon is gekozen.
  Door de juiste lijnen te trekken wordt een wave-patroon in een vlak gewist. Dit wordt gebruikt voor het tekenen van een nieuwe waarde in dit vlak.

- `drawMidWave :: MR VWindow -> Int -> Int -> Int -> IO()`
  Deze functie wordt gebruikt wanneer voor de "wave" representatie van het testpatroon is gekozen.
  Bij het tekenen van een nieuwe waarde in een wave-patroon is het nodig om te weten wat de waarden links en rechts van dit vlak zijn. Hiermee kan een mooi aaneengesloten patroon getekend worden. Hiervoor wordt de functie `drawMidWave` gebruikt. De functie heeft als een van de argumenten, het argument `WaveType`, een integer die aangeeft wat voor patroon er getekend moet worden. De codering is als volgt :
  - `0` ⇒ `000` ⇒ alles laag
  - `1` ⇒ `001` ⇒ links laag, midden laag en rechts hoog
  - `2` ⇒ `010` ⇒ links laag, midden hoog en rechts laag
  - `3` ⇒ `011` ⇒ links laag, midden hoog en rechts hoog
  - `4` ⇒ `100` ⇒ links hoog, midden laag en rechts laag
  - `5` ⇒ `101` ⇒ links hoog, midden laag en rechts hoog
  - `6` ⇒ `110` ⇒ links hoog, midden hoog en rechts laag
  - `7` ⇒ `111` ⇒ alles hoog

  Door de juiste lijnen te tekenen wordt zo een nieuwe waarde getekend.

- `drawHighWave :: MR VWindow -> Bool -> Bool -> Int -> Int -> IO()`
  Deze functie wordt gebruikt wanneer voor de "wave" representatie van het testpatroon is gekozen.
  Deze functie bepaalt de waarde voor het patroontype dat doorgegeven moet worden aan `drawMidWave` in het geval het middelste element hoog is. Dit kunnen de waarden 2,3,6,7 zijn.

- `drawLowWave :: MR VWindow -> Bool -> Bool -> Int -> Int -> IO()`
  Deze functie wordt gebruikt wanneer voor de "wave" representatie van het testpatroon is gekozen.
  Het tegenovergestelde van drawHighWave. De waarden kunnen dan 0,1,4,5 zijn.

- `draw_inputs :: MR Window -> MR VWindow -> [String] -> IO()`
  Met deze functie worden de invoerrijen in het invoergebied genummerd.

- `draw_output :: MR Window -> MR VWindow -> [String] -> IO()`
  Met deze functie worden de uitvoerrijen in het uitvoergebied genummerd.

- `draw_time :: MR Window -> IO()`
  Met deze functie worden de tijdstappen in het invoer- en uitvoergebied genummerd.

- `drawframe :: MR Window -> MR VWindow -> MR VWindow -> IO()`
  Deze functie wordt gebruikt wanneer voor de "wave" representatie van het testpatroon is gekozen.
  Met deze functie wordt het invoer- en uitvoergebied ingedeeld in vlakken. Hierbij wordt gewerkt met het lijntype `lineOnOffDash`.

- `drawgrid :: MR Window -> MR VWindow -> MR VWindow -> IO()`
  Deze functie wordt gebruikt wanneer voor de "block" representatie van het testpatroon is gekozen.
  Met deze functie wordt het invoer- en uitvoergebied ingedeeld in vlakken.

- `drawSquarePattern :: MR Window -> MR VWindow -> TestPattern -> IO()`
  Deze functie wordt gebruikt wanneer voor de "block" representatie van het testpatroon is gekozen.
  Met deze functie wordt een patroon afgebeeld. Voor alle invoerrijen en tijdstappen wordt met de functie `getAElem` uit de toestand de waarde gehaald van ieder vlak. Aan de hand van die waarde wordt de functie `black_square` of `white_square` aangeroepen.

- `drawWavePattern :: MR Window -> MR VWindow -> TestPattern -> IO()`
  Deze functie wordt gebruikt wanneer voor de "wave" representatie van het testpatroon is gekozen.
  Hetzelfde als de vorige functie, alleen nu voor een wave-patroon. Hiervoor worden ook nog de waarden van het linker en rechter vlak bepaalt.

- `drawBackGround :: MR Bool ->`
  `MR Window -> MR VWindow -> MR VWindow ->`
  `[String] -> [String] -> IO()`
  Deze functie maakt de "achtergrond". Dit wil zeggen dat de invoer- en uitvoergebieden voorzien worden van namen en worden ingevuld.

- `drawCarrier :: MR Bool -> MR Window -> MR VWindow -> MR VWindow ->`
  `IO()`
  Deze functie vult het invoer- en uitvoergebied al naar gelang de waarde van het eerste argumenten. Is dit True dan wordt voor de block-representatie gekozen. Anders voor de wave-representatie.

- `drawPattern :: TestPattern -> MR Bool ->`
  `MR Window -> MR VWindow -> MR VWindow -> IO()`
  Deze functie vult het patroon in al naar gelang de waarde van het tweede argumenten. Is dit True dan wordt voor de block-representatie gekozen. Anders voor de wave-representatie.

- `redrawPattern :: TestPattern -> MR Bool ->`
  `MR Window -> MR VWindow -> MR VWindow -> IO()`
  Met deze functie wordt het scherm opnieuw getekend.

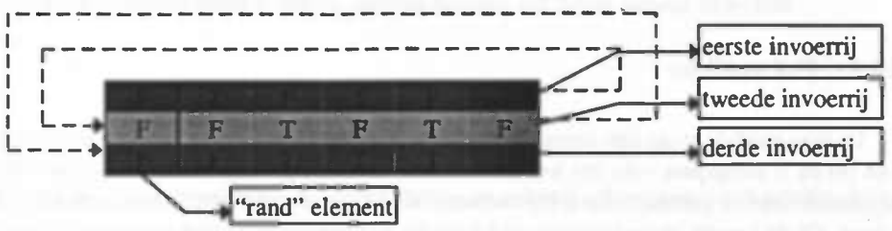
14.2.5 Patstate.hs

In deze module staan de functies die nodig zijn bij het aanmaken en beheren van de State Transformer die het testpatroon bevat. Het patroon wordt opgeslagen in een array van booleans die loopt van

`0 .. (#invoerrijen * (#aantal tijdstappen+1))`

De eerste invoerrij staat in de elementen `1 .. #aantal tijdstappen`.
De tweede invoerrij staat in de elementen `#aantal tijdstappen + 1 ... 2*num_t`, enzovoort. Er is tussen de twee rijen steeds een element extra dat altijd op False staat. Hiermee kan zonder nadenken steeds naar

een linker- of een rechter element gerefereerd worden, bijvoorbeeld bij het bepalen van het patroontype in drawHighWave en drawLowWave. Figuur 43 geeft een grafisch voorstelling.



Figuur 43 : Grafische voorstelling van de toestand array.

De functies uit deze module zijn :

- `getAElem :: TestPattern -> Int -> Int -> Int -> IO(Bool)`
  Met getAElem kan de waarde van een element uit de toestand gelezen worden. Hiervoor is nodig :
  - op welk tijdstip
  - op welke invoerrij
  - het aantal tijdstappen

  Hiermee kan de positie in de array berekent worden.

- `writeAElem :: TestPattern -> Int -> Int -> Int -> Bool -> IO()`
  Deze functie is het tegenovergestelde van getAElem.

- `getLines :: Int -> Int -> TestPattern -> IO[[Bool]]`
  De functie `getLines` wordt gebruikt door de functie `mkfunction` om een lijst met lijsten van booleans te krijgen. Dit is het tegenovergestelde van `fillPattern`.

- `getLine :: Int -> Int -> TestPattern -> IO[Bool]`
  De functie `getLine` wordt gebruikt door de functie `getLines` om een rij uit de toestand te halen. Deze functie is het tegenovergestelde van de functie `replaceLine`.

- `fillPattern :: [[Bool]] -> MR Window -> TestPattern -> IO()`
  De lijst met lijsten van booleans die de parser oplevert wordt met `fillPattern` in de toestand gezet. Hierbij kan het zijn dat de parser meer lijsten oplevert dan er in het invoergebied passen. Met de `take` functie wordt de lijst in dat geval ingekort tot de goede lengte.

- `replaceLine :: Int -> Int -> [Bool] -> TestPattern -> IO()`
  Deze functie wordt gebruikt door `fillPattern` om een lijst met booleans in de toestand op te slaan. Is de lijst langer dan het huidige aantal tijdstappen dan wordt de lijst in dat geval ingekort met de functie `take`.

- `getSubPattern :: MR Window -> TestPattern -> Int -> Int -> Int -> IO [Bool]`
  De functie `getSubPattern` haalt uit de toestand de lijst met te kopiëren elementen.

- `addSubPattern :: MR Window -> TestPattern -> Int -> Int -> [Bool] -> IO()`
  De functie `addSubPattern` wordt gebruikt bij het kopiëren van een deel van het patroon. Invoer is, onder andere, het nummer van de rij waarin moet worden gekopieerd, het tijdstip waarop begonnen moet worden met kopiëren in deze rij en de rij booleans die gekopieerd moet worden. Zou bij het kopiëren het tijdstip groter worden dan toegestaan dan wordt gestopt met kopiëren (de tak "`x == (num_t +1`")

- `resetPattern :: MR Window -> MR VWindow -> MR VWindow -> TestPattern -> MR Bool -> IO()`

Met deze functie wordt het patroon gereset. Iedere waarde wordt op false gezet.

14.2.6 Patwutil.hs

In deze module staan een aantal eenvoudige low-level grafische routines. Een groot aantal is overgenomen van [4] en is aangepast voor het werken met een variabele grootte van een window. Verder zijn er een aantal afgeleide functies gemaakt. Zo is bijvoorbeeld de functie `drawVertLine` gemaakt die een vertikale lijn tekent. Of de functie `drawVertUnitLine` die een vertikalelijn met een vaste[49] lengte tekent.

drawpoint	Int -> Int -> IO ()
drawline	Int -> Int -> Int -> Int -> IO ()
drawrectangle	Int -> Int -> Int -> Int -> IO ()
clearrectangle	Int -> Int -> Int -> Int -> IO ()
drawstring	Int -> Int -> String -> IO ()
draw_commands	MR Window -> [[Char]] -> Int -> IO()
endsession	MR Window -> MR Bool -> IO ()
clearscreen	MR Window -> IO ()
clClearAll	MR Window -> IO ()
clClear	MR Window -> Int -> Int -> IO ()
clcommandline	MR Window -> IO()
writeStatusBar	MR Window -> Int -> String -> IO()
clearStatusBar	MR Window -> IO()
clearError	MR Window -> IO()
clearArrea	Int -> Int -> Int -> Int -> IO ()
clWritePosition	MR Window -> Int -> Int -> String -> IO()
clWrite	MR Window -> Int -> String -> IO ()
errorstring	MR Window -> String -> IO ()
drawVertLine	Int -> Int -> Int -> IO()
drawHorzLine	Int -> Int -> Int -> IO()
drawVertUnitLine	Int -> Int -> Int -> IO()
drawHorzUnitLine	Int -> Int -> Int -> IO()
writeError	MR Window -> String -> IO()
getPoint	IO(Int,Int)
getUserClick	IO()

Tabel 39 : Functies uit de module **`Patwutil.hs`**.

[49] Deze lengte is ¼ van de lengte van een vlak

14.2.7 Commpatt.hs

- `patterncom :: MR DesignData -> Arguments -> IO()`

  Dit is de hoofdfunctie waarin bepaald wordt welke variant de gebruiker uitgevoerd wil hebben. De mogelijkheden zijn :
  - makeNeutralPattern
  - makeCurrentPattern
  - makeFileSysPattern
  - makeFileTuplePattern

  In § 14.1 staat uitgelegd wat deze verschillende varianten doen. Het aantal argumenten voor deze functies is variabel omdat optioneel gekozen kan worden voor het aantal invoerrijen, het aantal tijdstappen en de grootte van een vlak. In de desbetreffende functies wordt dit verder uitgezocht. In de functie `patterncom` wordt alleen de variant gekozen.

- ```
  makeNeutralPattern :: String -> Arguments -> IO()
  makeCurrentPattern :: MR DesignData -> PartName -> Arguments -> IO()
  makeFileSysPattern :: MR DesignData ->
                        FileName -> SysName -> Arguments-> IO()
  makeFileTuplePattern :: MR DesignData ->
                  FileName -> TupleName -> Arguments -> IO()
  ```

  Deze functies roepen de functie `patternMake` aan met de juiste argumenten. De waarden van deze argumenten hangen af van de invoer van de gebruiker. Wordt een bepaalde waarde niet gegeven door de gebruiker dan wordt de default waarde genomen. Wordt een fout argument gegeven, bijvoorbeeld een string in plaats van een geheel getal, dan wordt ook de default waarde genomen.

- ```
  patternMake :: FileName -> PartName ->
                                Int -> Int ->
                                [String] -> [String] -> IO()
  ```

  Dit is de initialisatie functie. De volgende acties worden uitgevoerd :
  - de gegevens van het omvattende window worden bepaalt en opgeslagen
  - de gegevens van het invoergebied worden bepaalt en opgeslagen
  - de gegevens van het uitvoergebied worden bepaalt en opgeslagen
  - een window wordt geopend
  - de menubalk wordt geopend
  - de array voor het testpatroon wordt aangemaakt
  - een drietal State Transformer booleans worden aangemaakt :
    - een voor het bepalen of de gebruiker wil stoppen met het programma
    - een voor het bepalen van welke representatie gebruik gemaakt moet worden. Is de waarde **`True`** dan moet de block-representatie gebruikt worden. Is de boolean **`False`**, dan moet de wave-representatie gebruikt worden.
    - een voor het bepalen of het testpatroon veranderd is sinds de laatste keer dat het is opgeslagen.
  - Hierna wordt gekeken of er al een file bestaat met de naam van het patroon. Bestaat deze file, dan wordt geprobeerd om die in te lezen.
  - Dan wordt de achtergrond getekend en wordt het patroon op het scherm gezet.
  - De hoofdlus, `makePattern` wordt nu aangeroepen, en, lekker imperatief, de controle wordt overgegeven aan `makePattern`. Als de gebruiker wil stoppen wordt doorgegaan met de functie `finalize`. Deze functie beëindigt het programma.

  In de where-clausules worden de juiste maten voor de verschillende windows berekend.

- `finalize :: MR Bool -> MR Window -> FileName -> PartName ->`

```
                                        TestPattern -> IO()
```

De functie `finalize` bepaalt aan de hand van het argument `changed` of het testpatroon veranderd is sinds de laatste keer dat het is opgeslagen. Is dit zo, dan wordt gevraagd of het patroon moet worden opgeslagen.

- `makePatName :: String -> String -> String`

  `makePatName` is een kleine functie die de filenaam maakt waarmee het patroon ingelezen of opgeslagen gaat worden.

- ```
  makepattern :: MR Bool -> MR Bool -> MR Bool ->
                 FileName -> PartName ->
                 MR Window -> MR VWindow -> MR VWindow ->
                 [String] -> [String] ->
                 TestPattern -> IO ()
  ```

  De functie `makepattern` is een lus waarin het volgende gedaan wordt :
  met de functie `getEvent` wordt een muisklik opgehaald. Er wordt gewacht, net zolang tot de gebruiker ergens aanklikt. Aan de hand van de positie van een muisklik wordt bepaald of :
  - de positie in de menu balk is. Voer dan de bijbehorende acties uit. Deze code spreekt voor zich.
  - de positie in het invoergebied is. Voer dan de bijbehorende acties uit. Deze code spreekt voor zich.
  - de positie ergens anders is. Doe dan niets.

- ```
  process_input :: MR Window ->
                   MR VWindow ->
                   TestPattern ->
                   MR Bool ->
                   MR Bool ->
                   (Int,Int) -> IO()
  flip_value :: MR VWindow -> MR Bool ->
                Bool -> Bool -> Bool ->
                Int -> Int -> IO()
  ```

  De functies `process_input` en `flip_value` zorgen samen voor het verwerken van een muisklik in het invoergebied. Een opmerking hierbij is dat hier ook de waarde van de boolean die bijhoudt of het patroon is veranderd, hier op True wordt gezet.

- ```
  copyPattern :: MR Window -> MR VWindow -> MR VWindow ->
                 MR Bool -> MR Bool ->
                 TestPattern -> IO()
   process_copy :: MR Window -> TestPattern ->
                   Int -> Int -> Int -> Int ->
                   IO(Bool,[Bool])
  ```

  Met deze twee functies wordt het kopiëren van een deel van het patroon uitgevoerd.
  ```

LITERATUUR LIJST

[1] J. Hughes.
"Why functional programming matters".
Report 16, Programming Methodology Group.
November 1984.

[2] J. McCharty.
"Towards a mathematical science of computation".
Information Processing 62. 1962. pp 22-28.

[3] The Aqua team.
'The Glorious Glasgow Haskell Compilation System. Version 0.23. User's guide".
Dec. 21, 1994.

[4] R. Krol.
"Functionele implementatie van grafische tools".
Intern rapport lab 9 project. November 18, 1994.

[5] R. T. Boute.
"Systems Semantics: Principles, Applications and Implementation".
In : ACM Transactions on Programming Languages and Systems, Vol. 10, Januari 1988, pp 118-155.

[6] A. Gordon.
"A Proposal for Monadic I/O in Haskell 1.3".
July 18, 1994.

[7] J. Launchbury, S. L. Peyton Jones.
"State in Haskell".

[8] M.P.Jones.
"Gofer release notes v2.30, juni 1994.

[9] P. Hudak, J.H. Fasel.
"A gentle Introduction to Haskell".

[10] J.T.W. Visser, H. Drenth.
"Gofer naar XNF vertaler". Intern rapport lab 9 project. May 1, 1995.

[11] R. Bird, P. Wadler.
"Introduction to functional programming".
Prentice Hall. 1988.

[12] C.N. Fischer, R. J. LeBlanc jr.
"Crafting a compiler"
Bejamin Cummings Publishing Company Inc. 1988

[13] S.L.Peyton Jones, D.R. Lester.
[Boek over Core Language]????

[14] ... Darlington ..... et al.

LIJST MET FIGUREN

FIGUUR 1 : DE ONTWERP-PRAKTIJK MET MEERDERE "GELIJKE" SYSTEMEN. 7
FIGUUR 2 : MEERDERE SEMANTISCHE MODELLEN VOOR ÉÉN BESCHRIJVING. 8
FIGUUR 3 : HET SYSTEEM CENTRAAL. 8
FIGUUR 4 : REPRESENTATIE VAN DE WAARDE VAN DE FUNCTIE STRUCT SEL. 9
FIGUUR 5 : HET RESULTAAT VAN DE APPLICATIE VAN SIMPLEX SEL. 9
FIGUUR 6 : HET SYSTEEM CENTRAAL. 10
FIGUUR 7 : VOORBEELD VAN HET ONTWERPEN MET TRANSFORMATIONAL DESIGN. 11
FIGUUR 8 : EEN GRAFISCHE WEERGAVE VAN STATE. 16
FIGUUR 9 : GRAFISCHE VOORSTELLING VAN DE FUNCTIE *THENST*. 16
FIGUUR 10 : EEN SAMENSTELLING. 17
FIGUUR 11 : EEN GRAFISCHE WEERGAVE VAN DE STATE TRANSFORMER *IO_STRING*. 17
FIGUUR 12 : GRAFISCHE VOORSTELLING VAN *NEWVAR*. 18
FIGUUR 13 : GRAFISCHE VOORSTELLING VAN *NEWVAR "DAG"*. 18
FIGUUR 14 : GRAFISCHE VOORSTELLING VAN *READVAR*. 18
FIGUUR 15 : GRAFISCHE VOORSTELLING VAN HET VOORBEELD. 19
FIGUUR 16 : GRAFISCHE VOORSTELLING VAN HET VOORBEELD. 19
FIGUUR 17 : GRAFISCHE VOORSTELLING VAN DE OPERATOR *THENST_*. 20
FIGUUR 18 : GRAFISCHE VOORSTELLING VAN DE OPERATOR *LISTST*. 21
FIGUUR 19 : GRAFISCHE VOORSTELLING VAN DE OPERATOR *LISTST_*. 21
FIGUUR 20 : GRAFISCHE VOORSTELLING VAN DE OPERATOR *MAPST*. 21
FIGUUR 21 : GRAFISCHE VOORSTELLING VAN DE OPERATOR *MAPST_*. 21
FIGUUR 22 : EEN GRAFISCHE WEERGAVE VAN DE STATE TRANSFORMER *IO_STRING*. 22
FIGUUR 23 : EEN GRAFISCHE WEERGAVE VAN DE FUNCTIE *PUTCHAR*. IN S' VAN HET TYPE *REALWORLD* IS S ZO 22
FIGUUR 24 : EEN GRAFISCHE WEERGAVE VAN DE FUNCTIE *GETCHAR*. IN S' VAN HET TYPE *REALWORLD* IS S ZO 22
FIGUUR 25 : GRAFISCHE VOORSTELLING VAN *USERANSWER* EN *READLN*. 23
FIGUUR 26 : EEN SCHEMATISCHE WEERGAVE VAN DE PLAATS VAN DE COMMANDOSHELL. 25
FIGUUR 27 : EEN VOORBEELD UIT DE SYSTEEM SUBSET. 33
FIGUUR 28 : LEXICAL OPBOUW VAN DE TOKENS. 34
FIGUUR 29 : DE OORSPRONKELIJKE GRAMMATICA VAN DE SYSTEEM SUBSET. 35
FIGUUR 30 : DE EQUIVALENTE GRAMMATICA WAARMEE GEWERKT GAAT WORDEN. 37
FIGUUR 31 : DE DATASTRUCTUUR VOOR DE INTERNE OPSLAG. 40
FIGUUR 32 : TEST DEFINITIE VOOR TABEL 11. 46
FIGUUR 33 : TEST DEFINITIE VOOR TABEL 11. 47
FIGUUR 34 : TEST DEFINITIE VOOR TABEL 12 47
FIGUUR 35 : VERGELIJKING MET EEN RELATIONELE DATABASE. 58
FIGUUR 36 : VOORBEELD VAN ANALYSE VAN COMMANDO VARIANT. 88
FIGUUR 37 : DE "ECHTE" INTERNE WEERGAVE VAN TWEE DEFINITIES. 91
FIGUUR 38 : HET RESULTAAT VAN HET COMMANDO "PA" TOEGEPAST OP DE STRUCTUUR UIT FIGUUR 37. 91
FIGUUR 39 : DE GRAMMATICA UIT VERSIE_1 95
FIGUUR 40 : HET INVOERGEBIED. 105
FIGUUR 41 : EEN VOORBEELD VAN DE UITVOER VAN *MKFUNCTION*. 110
FIGUUR 42 : DE GRAMMATICA VAN DE UITVOER. 110
FIGUUR 43 : GRAFISCHE VOORSTELLING VAN DE TOESTAND ARRAY. 114

LIJST MET TABELLEN

TABEL 1 : COMMANDO SET....26
TABEL 2 : SYNTAX VOOR KOSTEN COMMANDO....27
TABEL 3 : SYNTAX VOOR OVERZICHT COMMANDO....27
TABEL 4 : SYNTAX VOOR ENVIRONMENT COMMANDO....27
TABEL 5 : SYNTAX VOOR PRINT COMMANDO....27
TABEL 6 : SYNTAX VOOR HERHAAL COMMANDO....27
TABEL 7 : SYNTAX VOOR EINDE COMMANDO....28
TABEL 8 : OVERZICHT VAN DE GEBRUIKTE MODULES....30
TABEL 9 : ENKELE VERSCHILLEN TUSSEN GOFER EN DE GHC-COMPILER....31
TABEL 10 : FUNCTIES UIT DE MODULE SSLEXER.HS....46
TABEL 11 : VERGELIJKING VAN EVALUATIE KOSTEN VOOR HET SYSTEEM UIT FIGUUR 32....47
TABEL 12 : VERGELIJKING VAN EVALUATIE KOSTEN VOOR HET SYSTEEM UIT FIGUUR 33....47
TABEL 13 : VERGELIJKING VAN EVALUATIE KOSTEN VOOR HET SYSTEEM UIT FIGUUR 34....48
TABEL 14 : PROJECTIE FUNCTIE VOOR DE TYPE SYNONIEMEN VAN DE DATABASE....57
TABEL 15 : DE VERSCHILLENDE CATEGORIEËN DATABASE FUNCTIES TOEGEPAST OP DATABASE ONDERDEEL *ENV*....61
TABEL 16 : TYPERING VAN DE FUNCTIES VOOR HET DATABASE ONDERDEEL *ENV*....63
TABEL 17 : GEBRUIK EN BETEKENIS VAN DE FUNCTIES VOOR HET DATABASE ONDERDEEL *ENV*....63
TABEL 18 : TYPERING VAN DE FUNCTIES VOOR HET DATABASE ONDERDEEL *INFO*....63
TABEL 19 : GEBRUIK EN BETEKENIS VAN DE FUNCTIES VOOR HET DATABASE ONDERDEEL *INFO*....64
TABEL 20 : TYPERING VAN DE FUNCTIES VOOR HET DATABASE ONDERDEEL *SYSDEF*....64
TABEL 21 : GEBRUIK EN BETEKENIS VAN DE FUNCTIES VOOR HET DATABASE ONDERDEEL *SYSDEF*....65
TABEL 22 : TYPERING VAN DE FUNCTIES VOOR HET DATABASE ONDERDEEL *TUPLEDEF*....66
TABEL 23 : GEBRUIK EN BETEKENIS VAN DE FUNCTIES VOOR HET DATABASE ONDERDEEL *TUPLEDEF*....66
TABEL 24 : TYPERING VAN DE FUNCTIES VOOR HET DATABASE ONDERDEEL *BASEDEF*....67
TABEL 25 : GEBRUIK EN BETEKENIS VAN DE FUNCTIES VOOR HET DATABASE ONDERDEEL *BASEDEF*....68
TABEL 26 : TYPERING VAN DE FUNCTIES VOOR HET DATABASE ONDERDEEL *ATOMDEF*....68
TABEL 27 : GEBRUIK EN BETEKENIS VAN DE FUNCTIES VOOR HET DATABASE ONDERDEEL *ATOMDEF*....69
TABEL 28 : TYPERING VAN DE FUNCTIES VOOR HET DATABASE ONDERDEEL *COMMANDD*....69
TABEL 29 : GEBRUIK EN BETEKENIS VAN DE FUNCTIES VOOR HET DATABASE ONDERDEEL *COMMANDD*....69
TABEL 30 : TYPERING VAN DE FUNCTIES VOOR HET DATABASE ONDERDEEL *DEFAULTCOST*....70
TABEL 32 : TYPERING VAN DE FUNCTIES VOOR HET DATABASE ONDERDEEL *CURSYS*....71
TABEL 33 : GEBRUIK EN BETEKENIS VAN DE FUNCTIES VOOR HET DATABASE ONDERDEEL *CURSYS*....71
TABEL 34 : DE FUNCTIES UIT DE FILE COMMHGDB.HS....73
TABEL 35 : OVERZICHT VAN DE HELP FUNCTIES....74
TABEL 36 : DE VERSCHILLEN IN DE DATA STRUCTUUR MET OVEREENKOMENDE VARIANT....96
TABEL 37 : FUNCTIES VOOR HET DATATYPE *WINDOW*....108
TABEL 38 : FUNCTIES VOOR HET DATATYPE *VWINDOW*....109
TABEL 39 : FUNCTIES UIT DE MODULE PATWUTIL.HS....115