Een functionele aanpak voor taalcreatie & transformatie
Samenvatting

Softwareontwikkeling is een proces, waarin experts uit meerdere kennisdomeinen samen een applicatie ontwerpen en bouwen. In reguliere softwareontwikkeling sluit de oplossing niet altijd aan bij de voorafgestelde wensen van de klant. Dit kan komen doordat experts uit de verschillende kennisdomeinen niet begrijpen wat andere experts binnen het proces bedoelen. Deze onduidelijkheden kunnen ontstaan omdat de experts elk een eigen jargon gebruiken en deze jargons nauwelijks op elkaar aansluiten.

In deze scriptie wordt een methodiek voorgesteld die samen met een tool de onduidelijkheid tussen de domeinexperts uit verschillende kennisdomeinen binnen het softwareontwikkelingsproces kan verminderen. Het softwareontwikkelingsproces beperkt zicht tot de creatie van een informatiesysteem. De methodiek beschrijft een manier voor het definiëren van een isomorfe domeinabstractie voor elk betrokken domeingebied. De domeinabstracties worden geregerepresenteerd als natuurlijke talen. De tool, een taaltransformator, kan de verschillende domeinabstracties van en naar elkaar vertalen door middel van te definiëren transformaties. Om de verschillende abstracties naar elkaar te transformeren worden twee verschillende klassen van transformaties erkend: tussen tekst en syntaxisbomen en tussen twee syntaxisbomen. De eerste transformatievariant wordt beschreven met generisch programmeren. De tweede variant wordt gedefinieerd in Generalized Algebraic Data Types (GADTs) en wordt uitgevoerd door een evaluatiefunctie. Met behulp van deze transformaties kan kennis die beschreven is in een grammatica vertaald worden naar een andere isomorfe grammatica voor een ander kennisdomein. Hierdoor kunnen domeinexperts binnen het softwareontwikkelingsproces kennis tot zich nemen in hun eigen jargon en wordt de kans op onduidelijkheden kleiner.

De verschillende transformaties worden getoetst door het uitvoeren van transformaties op een simpele imperatieve programmeertaal en een domeinspecifieke taal.

Hoofdtermen:
Grammaticadefinitie, grammaticatransformatie, generisch programmeren, generalized algebraic data types (GADTs)
8 CONCLUSIE ........................................................................................................................................... 63

8.1 EVALUATIE ....................................................................................................................................... 63
  8.1.1 Grammaticacreatie .................................................................................................................... 63
  8.1.2 Grammaticatransformatie ......................................................................................................... 63

8.2 CONTRIBUTIE ................................................................................................................................... 66

8.3 VERVOLGONDERZOEK ..................................................................................................................... 66

APPENDIX A PARSER COMBINATORS .................................................................................................... 70

APPENDIX B TRANSFORMATIE VAN PROG VAR EXPR NAAR PROGC VARC EXPRC ...................... 72

APPENDIX C HULPPARSERS TRANSFORMATIETAAL ............................................................................ 75

APPENDIX D CHEETAH TAALDOCUMENTATIE ..................................................................................... 76

APPENDIX E TUSSENRESULTATEN TRANSFORMATIE WETTEKSTENAST NAAR WHILE_CST .......... 77
1 Introductie


Als dit voorgenoemde probleem wordt benaderd vanuit het probleemgebied, dan hebben de softwareontwikkelaars een ander beeld van het probleem en komen met een oplossing voor het beeld dat zij hebben van het probleem. Maar die oplossing lost misschien niet goed genoeg het oorspronkelijke probleem op. Wellicht sluit de oplossing niet goed aan bij het probleem of is deze zelfs incorrect of incompleet.

Hetzelfde doet zich voor bij de oplossing die door softwareontwikkelaars bedacht is. De oplossing is geschreven in een ander jargon dan van het probleemgebied, namelijk in een programmeertaal. Als er issues in de oplossing zijn, dan kunnen domeinexperts uit het probleemgebied nauwelijks achterhalen waardoor de issues ontstaan zijn. Dit komt omdat ze het jargon van de softwareontwikkelaars niet begrijpen. De oplossing is weliswaar expliciet gedefinieerd maar moeilijk of niet te begrijpen voor domeinexperts uit het probleemgebied.

Het probleem is samen te vatten in het beschikken over de benodigde kennis van het probleemgebied. Domeinexperts hebben onbewust meer kennis dan dat zij overdragen. Deze kennis zit verborgen in het jargon. Het stukje kennis, dat niet overgedragen wordt, speelt wel een rol bij de beoordeling van de oplossing. Door verschillende afwijkende interpretaties van kennis kunnen fouten ontstaan.

![Diagram](Figuer 1-1 - Communicatieprobleem verschillende domeingebieden)

Het communicatieprobleem wordt aan de hand van het voorbeeld uit Figuur 1-1 uitgelegd. Domeinexpert a heeft een probleem en defineert dit in zijn jargon (P_a). Domeinexpert b, een softwareontwikkelaar, begrijpt het jargon niet goed en creëert zijn perceptie van het probleem en zo ontstaat een tweede beeld op het probleem (P_b). De softwareontwikkelaar(s) gaan aan de slag en bouwen een oplossing in hun eigen jargon (O_b). Tot slot kijkt domeinexpert a naar O_b, maar het gebruikte jargon is voor hem onduidelijk en hij creëert zijn perceptie van de oplossing (O_a). Als O_a niet de oplossing is voor P_a, dan is willicht ergens in het communicatieproces iets mis gegaan.

---

1 Een issue is een probleem waarvan de oorzaak nog niet bekend is. Er kunnen verschillende oorzaken zijn, zoals specificatiefouten of implementatiefouten.
Er zijn verschillende manieren om dit probleem op te lossen. De verschillende partijen kunnen een zelfde jargon leren of er kan een vertaling gemaakt worden tussen twee jargons. Een simpele woord-voor-woord vertaling schiet soms tekort in de betekenis, zoals bij de uitdrukking “er is niets aan de hand”. Als deze uitdrukking letterlijk vertaald wordt naar het Engels, dan ontstaat de zin "there is nothing on the hand" voor een Engelsman iets heel anders betekent. Een vertaalslag tussen twee jargons kan op basis van de betekenis van de jargons tot stand gebracht worden. Hiervoor dient een abstractie gemaakt te worden van het jargon. Deze abstractie beschrijft de betekenis en de betekenis kan vervolgens worden vertaald. Het maken van een abstractie van een jargon kan worden gezien als het modelleren van het jargon.


In dit onderzoek wordt het jargon gemodelleerd in tekstuele modellen. De reden hiervoor is dat met tekstuele modellen een grote doelgroep te bereiken is, omdat ze in natuurlijke taal beschreven zijn. Daarnaast is het aantal regels in een tekstueel model vrijwel onbeperkt, waardoor grote oplossingen ook in het model te vangen zijn. De structuur van een tekstueel model, dat wil zeggen hoe instanties van het model eruit zien, wordt doorgaans vastgelegd in een grammatica. Andere manieren voor het vastleggen van de structuur van een tekstueel model zijn bijvoorbeeld eindige toestandsautomaten of Turingmachines. In dit onderzoek wordt de structuur vastgelegd in een grammatica. Deze grammatica moet het probleemgebied beschrijven. Maar voor welk domeingebied wordt de grammatica ontworpen? Vaak wordt de grammatica zo ontworpen dat deze dicht bij een bestaande natuurlijke taal of jargon ligt. De reden hiervoor is dat het op deze wijze minder tijd kost om de taal te ontwikkelen. Maar wel domeingebied begrijpt de taal dan?

De gebruikte taal moet aansluiten bij wat de gebruikers, de domeinexperts, begrijpen. In het softwareontwikkelingsproces zijn echter meerdere domeingebieden actief en dus ook meerdere domeinexperts. Met dit onderzoek wordt aangetoond dat verschillende domeingebieden tegelijkertijd samen een probleem kunnen oplossen door elk hun eigen taal of jargon te gebruiken, door te zorgen dat de verschillende talen naar elkaar vertaald kunnen worden.


---

Figuur 1-2 - Representaties XOR-poort

<table>
<thead>
<tr>
<th>Grafisch</th>
<th>Natuurlijke taal</th>
<th>Wiskundig</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="XOR-poort" /></td>
<td>C is waar als A ongelijk aan B is, in alle andere gevallen is C onwaar.</td>
<td>C = A \neq B</td>
</tr>
</tbody>
</table>

In dit onderzoek wordt het jargon gemodelleerd in tekstuele modellen. De reden hiervoor is dat met tekstuele modellen een grote doelgroep te bereiken is, omdat ze in natuurlijke taal beschreven zijn. Daarnaast is het aantal regels in een tekstueel model vrijwel onbeperkt, waardoor grote oplossingen ook in het model te vangen zijn. De structuur van een tekstueel model, dat wil zeggen hoe instanties van het model eruit zien, wordt doorgaans vastgelegd in een grammatica. Andere manieren voor het vastleggen van de structuur van een tekstueel model zijn bijvoorbeeld eindige toestandsautomaten of Turingmachines. In dit onderzoek wordt de structuur vastgelegd in een grammatica. Deze grammatica moet het probleemgebied beschrijven. Maar voor welk domeingebied wordt de grammatica ontworpen? Vaak wordt de grammatica zo ontworpen dat deze dicht bij een bestaande natuurlijke taal of jargon ligt. De reden hiervoor is dat het op deze wijze minder tijd kost om de taal te ontwikkelen. Maar wel domeingebied begrijpt de taal dan?

De gebruikte taal moet aansluiten bij wat de gebruikers, de domeinexperts, begrijpen. In het softwareontwikkelingsproces zijn echter meerdere domeingebieden actief en dus ook meerdere domeinexperts. Met dit onderzoek wordt aangetoond dat verschillende domeingebieden tegelijkertijd samen een probleem kunnen oplossen door elk hun eigen taal of jargon te gebruiken, door te zorgen dat de verschillende talen naar elkaar vertaald kunnen worden.


1.1 Aanleiding

Het communicatieprobleem zoals dat beschreven is in de vorige sectie is naar voren gekomen uit overleg met mijn afstudeerbegeleiders. Mijn onderzoek is inhoudelijk begeleid door Gert Veldhuijzen van Zanten en Betsy Pepels. De algemene begeleiding lag in handen van Jelle Gerbrandy en Theo Thijsse. De opdracht uitgevoerd is uitgevoerd voor de afdeling MDA Service, Model-Driven-Architecture afdeling, van Capgemini BAS B.V. De afdeling MDA Services gebruikt de Functional Model Driven Design (FMDD) methode voor de oplossing voor een door de klant aangedragen probleem. Hierin wordt een taal specifiek voor een klant domein gedefinieerd, die de domeinexperts dus begrijpen. De taal wordt vastgelegd in een grammatica en aan de hand van die grammatica kunnen de verschillende domeingebieden aan de slag.

Voor de domeinexperts is een gestructureerde tekstverwerker beschikbaar die op basis van de grammatica verschillende invoermogelijkheden beschikbaar stelt. Hiermee kan een domeinexpert zijn uitdrukking modelleren in de taal. Softwareontwikkelaars schrijven vervolgens transformatoren voor de grammatica, waarmee ze de uitdrukkingen naar een uitvoerbare oplossing vertalen. Voorbeelden van dergelijke oplossingen zijn C# of Java-code of een definitie voor een Business Process Management System (BPMS). De tool die ze daarvoor gebruiken en dit mogelijk maakt noemen ze Cheetah. De creatie van de taal en de transformatoren gaat moeilijker dan verwacht. Taalelementen worden nu met de hand gemaakt en daar komen veel technische details bij kijken. De volgorde van woorden wordt bijvoorbeeld handmatig bepaald en transformatoren worden zelfs specifiek voor woordsamenstellingen geschreven. De technische details weerhouden de softwareontwikkelaars van wat ze functioneel willen bereiken.

1.2 Probleemstelling

1.2.1 Vraagstelling

De belemmering bij de creatie van een domeinspecifieke taal is het startpunt van het onderzoek. In het onderzoek wordt het communicatieprobleem tussen verschillende domeingebieden in het softwareontwikkelingsproces aangepakt. In het bijzonder de communicatie tussen domeinexperts uit het probleem domein en informatici.

De hoofdvraag van het onderzoek is:

Hoe kan een verzameling jargon specifieke views op een probleemgebied vastgelegd worden, zodat een uitdrukking over het probleemgebied af te beelden is op de verschillende views en te vertalen is naar andere views?

Om de hoofdvraag te kunnen beantwoorden, moeten eerst twee deelvragen beantwoord worden. Ten eerste, hoe kan een jargon geformaliseerd worden naar een grammatica, zodat zinnen op basis van die grammatica begrijpelijk zijn voor de desbetreffende domeinexperts? En ten tweede, hoe kan een relatie tussen de verschillende jargons worden gelegd, zodat uitdrukkingen in een jargon te vertalen zijn naar een ander jargon?

Het doelgebied van deze scriptie beperkt zich tot het definiëren van een grammatica voor een taal voor de creatie van een definitie van een informatiesysteem. De creatie van de taal is een studie op zichzelf. Er is veel domein kennis nodig om een abstractie te kunnen maken van een domein en deze te beschrijven als een taal. Daarom richt de uitwerking van dit onderzoek zich niet op de taal zelf, maar op het formaliseren van een taal.

Bij de creatie van een informatiesysteem zijn zowel domeinexperts als softwareontwikkelaars betrokken die samen het informatiesysteem willen bouwen. Beide partijen weten hoe een dergelijk systeem in hun domein eruit ziet. Dat betekent echter niet automatisch dat ze van elkaar weten welk resultaat ze voor ogen hebben. De views van beide partijen beschrijven hetzelfde, alleen in hun eigen jargon.
Voor het onderzoek worden een tweetal aannamen gedaan namelijk: dat verschillende talen een isomorfe relatie kunnen hebben. Voor het onderzoek wordt een verzameling talen gebruikt waarvan sommige seminaatuurlijke talen zijn en één imperatieve programmeertaal. Door het isomorfisme en de vertalingen is de semantiek van de seminaatuurlijke talen impliciet. De tweede aannamer is dus dat de semantiek verstoort zit in de verschillende transformaties, maar niet alle transformaties hoeven semantiekbehouwend te zijn. De semantiek van de gebruikte imperatieve programmeertaal transformeert mee naar de natuurlijke taalrepresentaties.

1.2.2 Doelstelling

Het softwareontwikkelingsproces bestaat uit minimaal twee domeingebieden, waaronder in ieder geval het informaticadomein. Hierdoor ontstaan meerdere talen die het probleemgebied beschrijven. Deze talen hebben één ding gemeen, ze beschrijven hetzelfde probleemgebied, alleen vanuit een ander domeingebied. Het probleemgebied is de gemeenschappelijke kern van het proces en elk domeingebied heeft zijn eigen view daarop, zie Figuur 1-3. De views zijn isomorf, dat wil zeggen dat elke view een afbeelding van het probleem is en dat deze met behoud van structuur af te beelden is in een andere isomorfe view.

Niet alle views hoeven isomorf te zijn, zo kan een view gemaakt worden die maar een deel van het probleem weergeeft. Bijvoorbeeld een schooldomein waarin studenten en vakken beschreven zijn. Daarop kan een view worden beschreven die alle namen van studenten weergeeft die een vak volgen. Vanuit die view kan niet meer terug worden gegaan naar de oorspronkelijke vorm. Vanuit een niet isomorfe view kan niet meer terug worden vertaald.

De verschillende views, die ontstaan zijn door het formaliseren van de domeingebieden, worden op basis van hun betekenis naar elkaar vertaald, ofwel getransformeerd, zie Figuur 1-4. Dit kan omdat de verschillende jargons binnen het proces isomorf zijn. Hierdoor kan elk domeingebied het probleem in zijn eigen jargon of view bekijken en worden communicatiefouten verminderd.

Figuur 1-3 – Probleemgebied met meerdere domeingebieden

Figuur 1-4 - Domeinspecifieke views van een probleem

---

2 Isomorfisme is een bijectieve map tussen twee domeinen. Er is een functie \( f : : X \rightarrow Y \) waarvoor een inversie functie \( f^{-1} : : Y \rightarrow X \) bestaat waar zowel \( f \circ f^{-1} = id_Y \) als \( f^{-1} \circ f = id_X \) voor gelden.
Om het doel onderzoek te bereiken moet een verbetering in het softwareontwikkelingsproces gerealiseerd worden. Deze verbetering vindt plaats in de communicatie tussen de verschillende domeingebieden. Hierdoor begrijpen de verschillende domeinexperts beter wat anderen bedoelen en hierdoor wordt de kans op fouten kleiner. De kans op fouten wordt kleiner omdat een domeinexpert het probleem of de oplossing in zijn eigen jargon tot zich kan nemen en eventueel aan kan passen.

Naast de vermindering van communicatiefouten maakt het resultaat van dit onderzoek het mogelijk om views te verfijnen of te verbeteren. Bijvoorbeeld als domeinexpert b, uit Figuur 1-4, niet tevreden is met Viewb, dan kan een verfijnde Viewb' beschreven worden. Deze view beschrijft hetzelfde als Viewb, alleen met meer detail of op een andere manier. Op deze wijze kan een domeingebied zijn eigen view beheren en up-to-date houden. Door het up-to-date houden van het probleemgebied wordt, in het informaticagebied, de kans op legacy-systeem\(^3\) verkleind. Dit kan omdat er altijd een beschrijving van de oplossing is, waarop een nieuwe view gecreëerd kan worden. Als een vertaling tussen de oude naar de nieuwe view gemaakt wordt, dan kan de nieuwe view gebruikt worden in plaats van de oude, zie Figuur 1-5.

De verfijning en verbetering van views is een van de wensen van de afdeling MDA Services. Deze afdeling heeft als visie dat het systeem in zichzelf gedefinieerd moet kunnen worden en vanuit daar met hun eigen taal verfijnd en verbeterd kan worden. Daarnaast is het niet meer creëren van legacy-systeem een belangrijk punt in de verkoop van een nieuwe software.

### 1.3 Aanpak

Het vertalen van een taal naar een andere taal kan door middel van een compiler. Een compiler analyseert en parseert de invoer (tekst) en creëert daar een boomstructuur van. Deze boomstructuur geeft de zinstructuur weer. Daarna wordt op basis van de boomstructuur de uitvoer gecreëerd. Een compiler vertaalt één richting op, van de brontaal naar de doeltaal. Na compilatie is het meestal niet mogelijk om de doeltaal weer terug te vertalen naar de brontaal. Dit komt omdat structuurinformatie niet wordt opgeslagen. De inverse van de compiler is mogelijk door het maken van een andere compiler van de oorspronkelijke doeltaal naar de brontaal. Maar voor dit onderzoek moet tussen de verschillende talen, beide kanten open, vertaald worden en composites mogelijk zijn van verschillende deel vertaalstappen binnen één vertaalstuk, dus binnen één geheel. De traditionele compilaenoopak is dus niet geschikt als oplossing voor het communicatieprobleem.

Mijn begeleider, Gert van Veldhuijzen van Zanten, adviseerde mij om het probleem als een modelleerplichten te zien. In de modelleerwereld zijn verschillende views op modellen en transformaties tussen modellen standaard. Er is veel onderzoek gedaan naar modeltransformaties en dat onderzoek heeft zelfs geleid tot de standaard Query/View/Transformation (QVT) [23]. Als deze aanpak gebruikt wordt voor dit onderzoek, moet een relatie gelegd worden tussen de programmeerwereld en de modelleerwereld.

---

\(^3\) Een legacy-systeem, in deze context, is een softwaresysteem dat niet meer bij de tijd is. Dit kan verschillende oorzaken hebben zoals: verouderde technologie, achterstallig onderhoud of gebrek aan documentatie.
Er wordt gekozen om een modelleeraanpak te gebruiken voor dit probleem, de reden hiervoor staat beschreven in Sectie 1.3.1. Daarna wordt in Sectie 1.3.2 de keuze van de programmeertaal waar de taaltransformator in is gemaakt beschreven.

1.3.1 Keuze voor modelleeraanpak

De keuze voor de modelleeraanpak is gebaseerd op onderzoeksresultaten zoals [12, 17, 23]. In de modelleerwereld komen meerdere views op een model voor. Deze views en modellen worden beschreven binnen een metamodel-architectuur. In deze architectuur worden modellen gemodelleerd in een metamodel. Het metamodel is de beschrijving van het model. Op basis van metamodelen zijn transformaties naar andere modellen mogelijk. Er zijn verschillende modelltransformatieprogramma’s die een soortgelijke architectuur hebben [12, 17, 23].


Om een metamodel-architectuur te kunnen gebruiken voor dit onderzoek, moet een relatie worden gelegd tussen de programmeerwereld en de modelleerwereld. Als die relatie er is, dan kunnen de concepten uit de modelleerwereld gebruikt worden in de programmeerwereld en dus in dit onderzoek. In de programmeerwereld is een zelfde hiërarchie aanwezig als in Figuur 1-6. Hierin wordt dan niet gesproken over modellen maar over waardes en types. Een waarde is een instantie van een type en een type is gedefinieerd is een typedefinitie. Deze typedefinitie is op zijn beurt gedefinieerd in een taaldefinitie. Deze hiërarchie is weergegeven in Figuur 1-7.

Als we specifieker in de programmeerwereld naar de creatie van programmeertalen kijken, dan is er wederom een zelfde hiërarchie zichtbaar. Deze keer alleen over zinnen en grammatica’s. Een zin is gedefinieerd in zijn grammatica en een grammatica is gedefinieerd in de grammaticadefinitie. Deze grammaticadefinitie is zijn beurt gedefinieerd in een taaldefinitie. Deze hiërarchie is weergegeven in Figuur 1-8.

1.3.2 Keuze voor functionele programmeertalen

De taaltransformator kan geïmplementeerd worden in een programmeertaal. Hiervoor kunnen allerlei talen voor gebruik worden. De implementatietalen in dit onderzoek zijn functionele programmeertalen: Clean en Haskell. De eerste rede hiervoor komt vanuit het opdrachtvoorstel Capgemini BAS B.V., daarin is verwezen naar het functioneel programmeerparadigma. Daarnaast is de auteur van deze scriptie nieuwsgierig naar de mogelijkheden van functionele programmeertalen. In het vervolg van deze sectie wordt eerst dieper op deze twee beweegredenen ingegaan. Daarna worden de
De eerste beweegreden om functionele programmeertalen te gebruiken komt zoals gezegd voort uit de opdrachtomschrijving die door Capgemini BAS B.V. is voorgesteld. Daarin is gesproken over het functioneel programmeerparadigma voor het maken van een taaldefinitie. Mijn afstudeerbegeleiders wisten van mijn studieachtergrond in Nijmegen af en van mijn kennis over functionele programmeertalen. Zodoende is het functioneel programmeerparadigma naar voren gekomen. Dit is mijn eerste motieven geweest om te kijken of functionele programmeertalen inzetbaar zijn in een oplossing voor het communicatieprobleem.

De tweede beweegreden is explorerend van aard. Tijdens mijn studie in Nijmegen ben ik “geïnfecteerd” geraakt met het functioneel programmeervirus. Het zeer beknopt op kunnen schrijven van algoritmen, generiek programmeren en leuzen als “dat krijg je er voor niets bij” en “alles wat typecorrect is, is uitvoerbaar” hebben mij nieuwsgierig gemaakt naar de mogelijkheden van functionele programmeertalen. Ik wil exploreren wat functionele programmeertalen te bieden hebben en dit kan ik doen door ze in te zetten in mijn onderzoek.


Tot slot komt de keuze voor de gebruikte functionele programmeertalen aan de orde. In eerste instantie heb ik gebruik gemaakt van de functionele programmeertaal Clean. In een later stadium van mijn onderzoek is Haskell gebruikt om een deelprobleem op te lossen, voor de reden zie Sectie 4 en voor de toepassing zie Sectie 5.4. De verschillen tussen Clean en Haskell zijn qua syntaxis niet zo groot. Beide talen zijn gebaseerd op de lambda-calculus en de syntaxisverschillen tussen Clean en Haskell zijn minimaal [1]. Er zijn zelfs compilers die Haskell-syntaxis kunnen compileren naar Clean-syntaxis. Zoals de in Nijmegen zelf gebruikte beta-Clean-compiler, de “Haskell – Clean compiler” vanuit Eötvös Loránd University [7, 14] en het “Hacle project” vanuit de University of York [20]. Deze compilers laten zien dat de overstap van Clean naar Haskell klein is.

1.4 Testen aan de hand van een voorbeeld

Een van de beste manieren om te kijken of een vertaling mogelijk is, is door dit het daadwerkelijk uit te proberen. Daarom worden de vertalingen gedemonstreerd aan de hand van voorbeelden. In dit onderzoek wordt de While-taal [21] als voorbeeld gebruikt. Deze taal is een basis voor een imperatieve programmeertaal. De definitie is uitgebreid met een programma- en declaratiedeeldeel. Het voorbeeld dat gebruikt wordt is een natuurlijke taaldefinitie bovenop de betekenis van de While-taal. De specificatie van de While-taal is beschreven in Codevoorbeeld 3-6 en de natuurlijke taaldefinitie is beschreven in Codevoorbeeld 3-8.

1.5 Structuur scriptie

Het vervolg van deze scriptie is als volgt opgebouwd: in hoofdstuk 2 worden de verschillende talen die gebruikt worden in dit onderzoek toegelicht en welke klassen van talen er erkend worden. In dat hoofdstuk wordt beschreven hoe de grammatica van een taal beschreven kan worden in functionele programmeertalen. Het hoofdstuk eindigt met de relatie tussen de talen en modellen.
In hoofdstuk 3 wordt de scheiding beschreven tussen de essentie van een probleemgebied en de presentatie daarvan. Hiervoor worden twee grammatica’s beschreven: de abstracte grammatica voor de essentie en een (of meerdere) concrete grammatica’s voor de presentatie. Daarna wordt in hoofdstuk 4 de tool beschreven die taaltransformaties mogelijk maakt op basis van transformaties op grammaticaniveau. Vervolgens worden in hoofdstuk 5 de verschillende transformaties beschreven. Eerst wordt de pretty-printer beschreven die boomstructuren (grammatica-instanties) omzet naar tekst. Dan wordt de parser beschreven die tekst kan omzetten naar een boomstructuur. Daarna worden syntaxistransformaties beschreven tussen twee boomstructuren. Het hoofdstuk eindigt met een domeinspecifieke taal voor syntaxistransformaties. De beschreven transformaties uit hoofdstuk 5 worden getoetst door het uitwerken van een casus in hoofdstuk 6. Daarna worden in hoofdstuk 7 een aantal andere methodes en technieken getoond die mogelijk ook inzetbaar voor het verkleinen van het communicatieprobleem. Het laatste hoofdstuk van de scriptie is de conclusie. Dat hoofdstuk eindigt met een aantal interessante suggesties voor mogelijk vervolgonderzoek.
2 Talen en taalabstracties

In dit hoofdstuk wordt een introductie gegeven over de gebruikte varianten van taal. In deze scriptie wordt de term “taal” gebruikt voor het aanduiden van een tekstuele representatie van een domeinabstractie. De gebruikte talen zijn jargon-specifiek. Bijvoorbeeld een taal voor een informaticus is hetzelfde als de gebruikte programmertaal. Maar de taal voor een belastingexpert lijkt meer op een natuurlijke taal, hun jargon, dan op een programmertaal.

Alle isomorfe vertalingen van de verschillende talen behouden dezelfde semantiek. Dus als een taal genomen wordt waarvan de semantiek beschreven is dan krijgen alle isomorfe vertalingen die semantiek. Hieruit volgt dat een natuurlijke taalrepresentatie bovenop een programmertaal dezelfde semantiek krijgt wanneer daar een isomorfe vertaling voor wordt geschreven. Het behouden van de semantiek geldt voor alle isomorfe vertalingen en niet alleen voor programmertalen. Voor domeingebieden waar talen voor ontworpen worden, die sterk op natuurlijke talen lijken, wordt de term programmertaal gemeden. Deze term schrik t namelijk af door zijn technische achtergrond. Maar door de programmertaal te decoreren en te vertalen, kan deze gerepresenteerd worden als een seminatuurlijke taal. Hierdoor kunnen niet technische mensen, zonder de drempel van de term programmertaal, toch programmeren.

Dit hoofdstuk ziet er als volgt uit: Sectie 2.1 geeft een korte introductie over hoe talen en grammatica’s in deze scriptie worden gebruikt. In Sectie 2.2 wordt dieper ingegaan op het doel en uitdrukkingskracht van een taal. Na de beschrijving van wat talen zijn en waarvoor ze bedoeld zijn, gaat Sectie 2.3 over hoe talen vastgelegd kunnen worden in een functionele programmertaal. Tot slot wordt in Sectie 2.4 een beschrijving van de metamodel-architectuur gegeven.

2.1 Talen en grammatica’s

Om duidelijkheid te scheppen met wat bedoeld wordt met talen en grammatica’s, wordt eerst stil gestaan bij de definities van de woorden. Vervolgens worden deze definities gerelateerd met de manier waarop de woorden “taal” en “grammatica” in deze scriptie worden gebruikt.

De definities van taal en grammatica uit het Van Dale onlinewoordenboek [6]:

Taal spraakklanken waarmee men zijn gedachten en gevoelens aan anderen kenbaar maakt
Grammatica regels die beschrijven hoe een taal gesproken en geschreven wordt

Figuur 2-1 – Definitie van taal en grammatica uit het Van Dale onlinewoordenboek

Het Van Dale onlinewoordenboek gaat uit van natuurlijke taal. De talen in de context van dit onderzoek zijn minder vrij dan natuurlijke taal. Het betreft formele talen met een vooraf gedefinieerde grammatica en semantiek. Alle grammatica’s van dit onderzoek beschrijven tekst, maar een grammatica/model voor een grafische representatie is ook mogelijk.

Een taal wordt gebruikt om informatie over te dragen. Een grammatica beschrijft de structuurregels van een taal. Kort door de bocht genomen: het is voldoende om een grammatica te beschrijven om een taal zonder semantiek vast te leggen. Maar hoe wordt een grammatica vastgelegd, zodat deze kenbaar gemaakt kan worden voor anderen? Dit resulteert in een circulaire redenering: er is een taal nodig om een grammatica in te definieren. De taal die nodig is om de grammatica in te definieren kan men ook weer in een grammatica definieren. Om uit deze circulaire redenering te komen wordt een bestaande taal genomen, waarvan de grammatica al gedefinieerd is. In die taal worden de grammatica’s voor andere talen beschreven. In deze scriptie worden functionele programmertalen gebruikt om grammatica’s te definieren, zie Sectie 1.3.2.

Wanneer in deze scriptie over taal wordt gesproken, dan wordt het geheel om informatie vast te leggen en over te dragen bedoeld. Met de term grammatica wordt hetzelfde bedoeld als dat Van Dale (Figuur 2-1) doet: de regels waar een correcte zin in de taal aan voldoet. Hierbij moet wel gezegd
worden dat de zin alleen grammaticaal correct is. Dit wil nog niet zeggen dat er iets zinvol is. Of er iets zinvol is wordt door de semantiek van de taal bepaald. De gebruikte grammatica's binnen dit onderzoek laten grammaticaal correcte en semantisch incorrecte instanties toe en deze kunnen vertaald worden. Maar door het behoud van semantiek blijft de instantie van de doelgrammatica semantisch incorrect.

2.2 Generieke en domeinspecifieke talen

Talen kunnen verschillende domeinen kenbaar maken. Zo kunnen talen een groot of specifiek domein beschrijven. In deze sectie wordt eerst uitgelegd wat algemene talen (General Purpose Language, GPL) en domein specifieke talen (Domain Specific Language, DSL) zijn. Daarna wordt het verschil tussen GPLs en DSLs weergegeven. Tot slot wordt een beweegreden gegeven om tot het maken van DSLs over te gaan.

GPLs zijn talen die een groot probleemgebied bestrijken of beter gezegd het probleemgebied van een GPL overkoepelt alle probleemgebieden. Voorbeelden van GPLs zijn het Nederlands en Engels, daarnaast zijn er GPL programmeertalen zoals Clean, Haskell, C en de taal binnen de FMDD methode. Alle berekenbare functies zijn in deze talen uit te drukken, deze talen zijn Turing-compleet. De grammatica van een dergelijke taal is algemeen beschreven, zodat ook alles in de taal is uit te drukken. Een probleem moet uitdrukt worden in de grammatica van de GPL. Om een probleem precies uit te drukken in een GPL zijn vaak veel woorden en grammaticaregels nodig om de semantiek van de uitdrukking precies vast te leggen.

DSLs zijn talen die een specifiek probleemgebied bestrijken. Bijvoorbeeld de taal SQL die de taal over relationele databases beschrijft. DSLs zijn bedoeld om een domein duidelijk en beknopt te beschrijven. DSLs kunnen soms niet alles uitdrukken, ze zijn niet altijd Turing-compleet. Soms worden DSLs vervuld en Turing-compleet gemaakt om net iets meer uit te drukken dan het oorspronkelijke probleem. Het domeinspecifieke van een DSL zit in de eenvoud waarmee een domeinspecifiek probleem te beschrijven is. Problemen uit het domeingebied laten zich goed tot uitstekend beschrijven in de bijbehorende DSL. Echter valt het probleem buiten het domein van de DSL, dan is de DSL minder of helemaal niet geschikt. Hiervoor kan wel een GPL gebruikt worden.

De uitdrukking in een DSL is veelal duidelijker voor het desbetreffende domeingebied dan dezelfde uitdrukking in een GPL. Dit komt doordat woorden in een DSL meer zeggen dan in een GPL. De semantiek van de DSL ligt vast in de grammatica. Hierdoor wordt het maken van kleine slordigheid fouten verklein, zoals het verkeerd plaatsen van haakjes in een formule of de volgorde van bewerking. In een DSL wordt de semantiek bepaald door de taalelementen en niet door de inhoud van de taal. DSLs kunnen het probleemgebied leesbaarder, duidelijker en beknopter beschrijven dan een GPL.

In deze scriptie wordt echter geen onderscheid gemaakt tussen GPLs en DSLs. Het gaat immers om de grammatica van de taal en niet om de uitdrukking van de taal. Talen worden van en naar elkaar getransformeerd op basis van de grammatica en niet op basis van een specifieke uitdrukking in een taal, er wordt niet getransformeerd op basis van de inhoud van de grammatica. Het maakt voor de toepassing van dit onderzoek wat er wel meer raakvlak hebben met het gebied van DSLs, omdat deze technologie het maken en onderhouden van DSLs makkelijker maakt.

2.3 Relatie types en grammatica's

In deze sectie wordt de aandacht gevestigd op de relatie tussen type en grammatica. De aandacht wordt hierop gevestigd omdat types erg belangrijk zijn in de gebruikte sterk getypte functionele programmeertalen. In een dergelijke taal heeft namelijk elk taalelement of uitdrukking een type. Er is niet aan gebruik van types te ontkomen in een sterk getypte functionele programmeertaal. In Sectie 2.3.1 wordt uitgelegd wat types zijn en welke smaken types er zijn. Daarnaast wordt in die sectie ook even stilgestaan bij twee verschillende schrijfwijzen van datatypes. Tot slot wordt in sectie 2.3.2 uitgelegd wat het type van een type is.
2.3.1 Types in een functionele programmeertaal

2.3.1.1 Datatypes


```
i = 0   // i :: Int
s = "Bjorn"  // s :: String      == array van karakters
xs = [1,2,3]  // xs :: [Int]      == lijst van integers
t = (xs, True)  // t :: ([Int], Bool)     == een tuple
r = { naam = "S" }  // r :: R, waarbij :: R = { naam :: String } == een record
```

Codevoorbeeld 2-1

De bovengenoemde representaties kunnen in principe alle mogelijke combinaties van data uitdrukken, maar niet altijd even eenvoudig. Zowel Clean als Haskell heeft de mogelijkheid om Algebraïsche Data Types te definiëren. Algebraïsche Datatypes zullen vanaf nu ADTs worden genoemd. Een ADT is een datatype dat één of meerdere dataconstructoren heeft. Elke dataconstructor heeft nul of meer argumenten, deze argumenten kunnen van elk willekeurig type zijn. ADTs worden uitgelegd aan de hand van een voorbeeld.

```
:: Persoon  = Persoon String Int
:: Weekdag  = Zo | Ma | Di | Wo | Do | Vr | Za
:: List a   = Cons a (List a) | Nil
```

Codevoorbeeld 2-2

In Codevoorbeeld 2-2 zijn drie ADTs te zien, namelijk Persoon, Weekdag en List a. Deze drie ADTs hebben samen tien dataconstructoren: Persoon, de eerste twee letters van de zeven weekdagen, Cons en Nil. Het Persoon-datatype heeft één dataconstructor met een ariteit van twee, dat wil zeggen: heeft twee argumenten nodig. Het Weekdag-datatype heeft zeven dataconstructoren, alle zeven met ariteit nul. De verschillende dataconstructoren worden gescheiden voor een |-teken. In het List-voorbeeld is een geparameteriseerd datatype te zien. In dat voorbeeld kan de parameter a van elk willekeurig type zijn, bijvoorbeeld een List Int of iets complex als (List Char, List Persoon). Daarnaast roept de dataconstructor Cons zijn type recursief aan, de dataconstructor Cons heeft als tweede parameter een List a. Een instantie van een ADT is altijd één van de dataconstructoren van de ADT.

ADTs kunnen herschreven worden naar een combinatie van de basisstructuren uit Codevoorbeeld 2-1. Zo zijn bomen bijvoorbeeld uit te drukken in tupels. Alleen beschrijft de ADT de boom preciezer en met meer betekenis. Zie Codevoorbeeld 2-3, daarin wordt een boomnotatie weergegeven in geneste tupels. Daarnaast is het schrijven van functies voor basisstructuren die een complexer type representeren moeilijk omdat de structuur behouden moet worden.

```
:: Tree a    = Node (Tree a) (Tree a) | Leaf a
tree   = Node (Node (Leaf 3) (Leaf 2)) (Leaf 1)
```

Codevoorbeeld 2-3

De grammatica voor het maken van ADTs is LL(1), parserend van links naar rechts, afleidend vanuit de linkerkant en er hoeft maar één teken (token) gelezen te worden om het type te identificeren. Dit komt omdat binnen één programma dataconstructoren uniek moeten zijn. Hieruit volgt dat als een dataconstructor we legen gekomen wordt, altijd met zekerheid gezegd kan worden dat er met het type Weekdag gewerkt wordt. Ook kan gezegd worden dat na het lezen van de dataconstructor Cons altijd met een type List a gewerkt wordt (met dan nog een willekeurige a) en dat er dan nog twee argumenten komen, namelijk de a die het type van a bepaalt en een List a. Aan de hand van de grammatica voor ADTs kunnen complexe datastructuren gebouwd worden. In dit onderzoek zijn ADTs gebruikt om grammatica’s vast te leggen. De grammatica wordt vastgelegd door woorden en woordsamenstellingen als ADTs te definiëren en daar een semantiek aan te geven.
Naast het gebruik van ADTs, is in dit onderzoek ook gebruik gemaakt van Generalized Algebraic Data Types (GADTs). Waar bij ADTs eerst het type wordt geven, gevolgd door dataconstructoren met argumenten, wordt bij een GADT eerst de dataconstructor gegeven gevolgd door eventuele argumenten en dan het type. Binnen een GADT is het mogelijk het type van variabelen open te laten, dat wil zeggen dat bij het definiëren van een type typeparameters open gelaten kunnen worden en deze door de type-instantie bepaald kunnen worden. In Haskell worden deze variabelen “rigid” genoemd. Alle dataconstructoren van een GADT worden gevangen in een soort van klassesdefinitie van het type.

```haskell
data Persoon where
    Persoon :: String -> Int -> Persoon

data Weekdag where
    Zo :: Weekdag
    Ma :: Weekdag
    Di :: Weekdag
    Do :: Weekdag
    Vo :: Weekdag
    Vr :: Weekdag
    Za :: Weekdag

data List a where
    Cons :: a -> (List a) -> List a
    Nil :: List a
```

Codevoorbeeld 2-4

In het bovenstaande codevoorbeeld zijn de GADT representaties van de ADTs uit Codevoorbeeld 2-2 te zien. In dit geval maakt het niet uit of voor ADTs of GADTs gekozen wordt. De Glasgow Haskell Compiler (GHC) kan bepalen of een typedeclaratie als ADT te schrijven is. Niet alle GADTs zijn als ADT te schrijven, bijvoorbeeld GADTs met “rigid”-variabelen. Deze types vallen buiten de Haskell-98 standaard. Op datatypes die buiten de Haskell-98 standaard vallen kunnen geen standaard klassen en generische functies voor worden afgeleid, zoals `Show` en `Eq`.

```haskell
data Term a where
    Lit :: Int -> Term Int
    Inc :: Term Int -> Term Int
    Isz :: Term Int -> Term Bool
    If :: Term Bool -> Term a -> Term a -> Term a
    Pair :: Term a -> Term b -> Term (a, b)
    Fst :: Term (a, b) -> Term a
    Snd :: Term (a, b) -> Term b
```

Codevoorbeeld 2-5

Het bovenstaande codevoorbeeld, uit [24], toont een veelgebruikt voorbeeld om GADTs te demonstreren. In het voorbeeld is te zien dat dataconstructoren het type van de parameter `a` vastleggen en dat dit niet meer bij de typedeclaratie gebeurt. Zo levert de dataconstructor `Lit` een `Term Int` op en `Pair` een `Term (a, b)`. Bij een GADT bepaalt de dataconstructor het geparameteriseerde type. Als deze declaratie door middel van een ADT was gemaakt, dan had elke dataconstructor van hetzelfde type moeten opleveren (dezelfde `a`) en dat kan niet in dit voorbeeld. Het type `Term a` kan niet als ADT geschreven worden. Voor het type `Term a` kunnen geen standaard klassen zoals `Show` en `Eq` voor worden afgeleid.

ADTs en GADTs zijn geschikt voor het vastleggen vast leggen van structuur: uit welke elementen bestaat het datatype. Echter de datatypes zijn niet heel precies over welke informatie vastgelegd wordt. Zo is het bijvoorbeeld niet mogelijk om een datatype te creëren dat alle even getallen tussen 0 en 100 representeert. In talen als Agda [22] en Epigram [19] zijn dependent-types te definiëren, waarin precies aangegeven kan worden welke elementen binnen het typen vallen. Dus in die talen kan wel een type gecreëerd worden dat alle even getallen tussen 0 en 100 representeert. In de gebruikte talen van dit onderzoek, Clean en Haskell, kan dat niet. Voor de taal waarin de taaltransformator is geïmplementeerd is niet gekozen voor een taal die dependent-types toelaat. Dependent-types zijn te complex voor het vastleggen van grammatica’s. Dit komt door de expressie die het type kan beperken, de expressie kan niet termineren en daardoor onbeslisbaar zijn. Alle extra complexiteit die erbij komt bij het gebruikt van dependent-types valt buiten de scope van dit onderzoek. In vervolg onderzoek kunnen de gebruikte ADTs en GADTs naar het dependent-types-domein gelift worden.
### 2.3.1.2 Functietypes

Datatypes representeren een waarde en doen verder niets. Functies herschrijven instanties van types en hebben een brontype en doeltype. Functietypes worden aangegeven door een pijlinnotatie (\( \rightarrow \)) en moeten gelezen worden als “van ... naar ...”. De functie-instantie bepaalt wat precies met de broninstanties gebeurd. Het schrijven van een functietype verschilt tussen Clean en Haskell. In Clean hoeven geen functiepijlen tussen de argumenten van de functie, alleen tussen de argumenten en het resultaattype, en in Haskell wel (zie Codevoorbeeld 2-6).

```plaintext
// In Clean
plus :: Int Int -> Int

// In Haskell
plus :: Int -> Int -> Int
```

In de GADT specificatie, uit Codevoorbeeld 2-5, voor het type `Term a` is te zien dat de dataconstructor `Lit` van het type `Int -> Term Int` is. Dus `Lit` is een functie van `Int` naar `Term Int`, gegeven een `Int`-waarde levert deze functie een `Term Int` op.

Een functie-instantie kan uit meerdere regels bestaan en de eerst passende regel wordt uitgevoerd, er wordt een first-fit-strategie gebruikt. Met behulp van pattern-matching kan de invoer worden gecontroleerd op de juiste instanties. Controles kunnen worden gedaan op de inhoud van de brontypes, zoals op elementen in een lijst of op dataconstructoren. Door pattern-matching kan toegang verkregen worden tot de argumenten van de instantie. In Codevoorbeeld 2-7 is te zien dat door te pattern-matchen op de dataconstructor `Assign` toegang tot de argumenten toegang verkregen wordt.

```plaintext
:: Stmt v e  = Assign v e
:: StmtC v e   = AssignC v Space AssignSign Space e
trans :: (Stmt v e) -> (StmtC vC eC) | trans v vC & trans e eC
trans (Assign v e)  = AssignC (trans v) Space AssignSign Space (trans e)
```

### 2.3.2 Kind: het type van types

In de vorige sectie is gesproken over types en instanties van die types. Wanneer types en instanties in de context van een taalhiërarchie geplaatst worden, waar zijn types dan in uitgedrukt? Het type van een type is een Kind (als Engelse term). Eerst wordt uitlegge wat Kinds zijn en tot slot wordt uitgelegd wat kinds te maken hebben met dit onderzoek.

Kinds weerspiegelen de ariteit van types, hoeveel argumenten en van welk Kind er nodig zijn om een instantie te krijgen van het doeltype. Een Kind wordt aangegeven als een ster of een functie van Kinds, zie de grammatica voor Kinds in Codevoorbeeld 2-8.

```
Kind ::= *
   | (Kind -> Kind)
```

![Figuur 2-2 - Het kind van een type](image)

Kinds worden uitgelegd aan de hand van het voorbeeld in Figuur 2-2. Ingevulde datatypes zijn altijd van Kind `*`, zowel standaard datatypes als samengestelde datatypes. Functies hebben altijd een
argument, anders was het geen functie, en zijn daarom altijd van de vorm \((\text{Kind} \rightarrow \text{Kind})\). Opmerkelijk is dat wanneer een dataconstructor zonder argumenten wordt gebruikt een functietype ontstaat, van de argumenten van de dataconstructor naar het type. De implementatie van die functie zit verstopt in het datatype. Het `assign`-voorbeeld uit Figuur 2-2 heeft enige gelijkenis met de GADT-notatie, in de typedeclaratie komen eerst de argumenten en dat het type. De `Kind` van een type zegt wat met een instantie van dat type mogelijk is. Een `Kind` * is data. Op data een kan pattern-match worden gedaan. Elke `Kind` anders dan *, is een functie en zal de argumenten herschrijven.

Waarom zijn `Kinds` nu van toepassing op dit onderzoek? In dit onderzoek wordt de dataconstructor gebruikt om een uitdrukking in de taal kenbaar te maken. Deze uitdrukkingen worden naar een andere taal getransformeerd en ook daarin wordt de uitdrukking in een combinatie van een dataconstructor en argumenten opgeslagen. In de transformatie wordt het doeltype opgebouwd door middel van stappsgewijze functiecompositie. `Kinds` spelen binnen dit onderzoek een rol bij de creatie en de transformatie van de grammatica’s omdat naar grammatica-instanties van `Kind` * gewerkt wordt. Daarnaast spelen `Kinds` een belangrijke rol in het generisch raamwerk van Clean, in de generische wereld is er een klasse voor elk `Kind`.

### 2.4 Beschrijving metamodel-architectuur

In deze sectie wordt de metamodel-architectuur nader beschreven. Eerder, in Sectie 1.3.1 is de relatie gelegd tussen de programmeerwereld en de modelleerwereld. Hier wordt de architectuur uit de modelleerwereld in de context van de programmeerwereld geplaatst.

In de figuren 1-6, 1-7 en 1-8 zijn de drie modellen en de instantielagen te zien, in de laatste twee figuren in de programmeercontext. Elke laag in de metamodel-architectuur heeft een wederhelft in de programmeerwereld. Zo staan zinnen in een grammatica gelijk aan modelinstanties in een model. In Figuur 2-3 is de relatie tussen de programmeerwereld en modelleerwereld schematisch weergegeven.

<table>
<thead>
<tr>
<th>Modelleerwereld</th>
<th>Programmeerwereld</th>
<th>Kenobject</th>
<th>EBNF</th>
</tr>
</thead>
<tbody>
<tr>
<td>M3 Meta-metamodel</td>
<td>Types/Grammatica van de programmeertaal</td>
<td>Clean</td>
<td>Transformatietaalgrammatica</td>
</tr>
<tr>
<td>M2 Metamodel</td>
<td>Programmeertaal</td>
<td></td>
<td>Transformeer a door</td>
</tr>
<tr>
<td>M1 Model</td>
<td>Grammatica domein specifieke taal</td>
<td></td>
<td>Als a = x dan x’</td>
</tr>
<tr>
<td>M0 Model instantie</td>
<td>Zinnen/statements</td>
<td></td>
<td>Als a = y dan y’</td>
</tr>
</tbody>
</table>

**Figuur 2-3 – Relatie modelleerwereld en programmeerwereld**

Naast de beschrijving van modellen heeft een metamodel-architectuur ruimte voor modellen die de transformatie tussen modellen beschrijven, het transformatiemodel. Daarbij hoort ook de beschrijving van dat transformatiemodel. Ook dit transformatiemodel en bijbehorend metamodel kunnen worden gezien als grammatica’s. Zo kan gezegd worden dat een grammatica gemaakt kan worden die de transformatie tussen twee andere grammatica’s beschrijft. Met het bronmodel, het doelmodel en het transformatiemodel ontstaan drie kolommen in de architectuur. Wanneer de drie modellen en drie kolommen ingevuld worden met de gegevens van dit onderzoek, dan ziet de metamodel-architectuur eruit als in Figuur 2-4.
Met behulp van deze architectuur en een executiefunctie kunnen instanties, zinnen die voldoen aan een grammatica, aan de hand van een transformatie-instantie vertaald worden naar een instantie van een andere grammatica. Bijvoorbeeld een vertaling van het “geformaliseerde Nederlands”\(^4\) naar het “geformaliseerde Engels”\(^4\). Daarbij zijn eerst grammatica’s voor de Nederlandse en Engelse taal en een transformatiegrammatica nodig. In de transformatiegrammatica wordt op basis van de grammatica’s een instantie geschreven die de vertaling van het Nederlands naar het Engels beschrijft. Met behulp van de executiefunctie kunnen zinnen, die voldoen aan geformaliseerde Nederlandse grammatica, samen met de eerder genoemde transformatie-instantie vertaald worden naar het geformaliseerde Engels. In Figuur 2-5 is deze vertaling in pseudocode weergegeven.

\[
\text{transformatie} :: (\text{GrammTrans Nederlands Engels}) \rightarrow \text{GrammEngels}
\]
\[
\text{transformatie trans ned} = \text{maakTransformatie trans ned}
\]

\[
\text{maakTransformatie} :: (\text{GrammTrans a b}) \rightarrow (a \rightarrow b)
\]

Figuur 2-5 – Transformatie-executie in pseudocode

Op basis van deze architectuur, samen met de ADTs en GADTs voor de grammatica, wordt de taaltransformator gebouwd in functionele programmeertalen.

\(^4\) Het geformaliseerde geeft aan dat het gedeelte van de taal eenvoudig en eenduidig in een grammatica vast te leggen is. Het geformaliseerde Nederlands beschrijft een deel van de Nederlandse taal.
3 Vormen van kennisrepresentatie

Kennis kan op verschillende manieren geregistreerd en vastgelegd worden. In dit onderzoek wordt kennis gepresenteerd als een taal vastgelegd in een grammatica. In dit onderzoek worden twee soorten grammatica's om kennis vast te leggen erkend: abstracte en concrete grammatica's.

Metalalen zoals (E)BNF zijn bedoeld om syntaxis vast te leggen. Maar wat is de betekenis van de volgende uitdrukking: "variabele = 1"? Is het een toekenning, een vergelijking of wordt het bepaald door de context waarin de uitdrukking staat? In abstracte syntaxis wordt onderscheid gemaakt tussen een toekenning of een vergelijking door dit expliciet te vermelden. De zin kan in abstracte syntaxis als vergelijking gedefinieerd worden, als ADT, in de vorm `Compare (Var "variabele") 1` en als toekenning in de vorm `Assign (Var "variabele") 1`. In deze representatie wordt de betekenis van de zin bepaald door de woorden `Compare` en `Assign`. In concrete syntaxis moet de zin de semantiek weerspiegelen. De betekenis van de concrete uitdrukking "variabele = 1" is niet precies duidelijk en mogelijk ambigu. Voor de concrete syntaxis moet ambiguïteit worden vermeden.

Grammatica's in deze scriptie zijn beschreven in ADTs of GADTs. Deze datatypes lijken sterk op (E)BNF en zijn wat met herschrijven werk eenvoudig in die vorm te schrijven. De grammaticadefinities moeten duidelijk zijn voor iedere die (E)BNF grammatica's kan lezen. De dataconstructoren zijn de terminals en types zijn de non-terminals.

In de rest van dit hoofdstuk wordt uitgelegd wat de abstract en concrete syntaxis is en wat de verschillen zijn. Dit gebeurt aan de hand van een voorbeeldprogramma dat de faculteit berekend, zie Codevoorbeeld 3-1. Vervolgens wordt de While-taal beschreven, zowel in abstracte als concrete syntaxis. Deze taal wordt in het vervolg van de scriptie gebruikt in de voorbeelden.

Programma[val]
Proc calcFactorial (x)
Begin
  y := 1
  Zolang x>1 Doe
    y := y * x
    x := x - 1
  End
End
Begin
  result := Voer uit calcFactorial(val)
End
Codevoorbeeld 3-1

3.1 Abstracte syntaxis

Abstracte syntaxis abstraheert van representatie. De abstracte syntaxis kan worden gezien als een abstracte syntasisboom (Abstract Syntax Tree, AST). De elementen van de syntaxis bepalen de semantiek. Neem bijvoorbeeld het taaltje IntExpr uit Codevoorbeeld 3-2 en de expressies zes, zes' en acht, de betekenis van die expressies is de formule die achter de expressie staat. In de abstracte syntaxis zijn geen karakters of woorden te vinden die de betekenis van de expressie doet veranderen. Alle haakjes in de expressies zijn bedoeld voor expliciete parameteraanduiding van de dataconstructoren en niet om de semantiek van de expressie te beïnvloeden. De expressies zes en zes' zijn qua syntaxis en AST niet aan elkaar gelijk, maar de uitkomst na berekening wel.
De lambda-calculus is een wiskundige taal met weinig decorerende taalelementen. Deze taal laat zich eenvoudig omzetten in een grammatica voor de abstracte syntaxis, deze is te zien in Codevoorbeeld 3-3. Lambda-expressies kunnen in deze taal gemodeleerd worden met veel overeenkomsten met de normale schrijfwijze.

```
:: LambdaExpr = Apply LambdaExpr LambdaExpr
| Lambda NVariabele LambdaExpr
| If LambdaExpr LambdaExpr LambdaExpr
| LVariable NVariabele
| Value NValue

:: NVariabele = Variable String
:: NValue = Integer Int
| Bool Bool
```

Codevoorbeeld 3-3

SUCC := \(\lambda \, n \, f \, x \cdot f \, (n \, f \, x)\)
SUCC\_lambdaExpr = Lambda (Variable “n”) (Lambda (Variable “f”) (Lambda (Variable “x”) (Apply (LVariable (Variable “f”)) (Apply (Apply (LVariable (Variable “n”)) (LVariable (Variable “f”))) (LVariable (Variable “x”))))))

Codevoorbeeld 3-4

In Codevoorbeeld 3-4 is de successor-functie in zowel lambda-calculus als LambdaExpr weergegeven. De laatste is moeilijk leesbaar, maar de semantiek is precies gevangen in de AST. Als SUCC\_lambdaExpr opgepoetst wordt door: het woord Lambda te vervangen door \(\lambda\), alle haakjes te verwijderen behalve om de node Apply, alle aanhalingstekens te verwijderen en de woorden Variable, LVariable en Apply te verwijderen, dan ontstaat een expressie zoals beschreven staat in Codevoorbeeld 3-5. De abstracte syntaxis is misschien onleesbaar, maar weerspiegelt wel precies de betekenis.

SUCC\_lambdaExpr = \(\lambda \, n \cdot \lambda \, f \cdot \lambda \, x \cdot f \, ((n \, f) \, x)\)

Codevoorbeeld 3-5

Een zelfde soort grammatica, als voor de lambda-calculus, kan ook voor een eenvoudige imperatieve programmeertaal worden gemaakt. Als basis hiervoor wordt de While-taal uit [21] gebruikt, na uitbreiding met een programmagedeelte en declaratiedeeldeel. Hierdoor kunnen procedures en procedure-aanroepen ook in de taal worden opgenomen. De abstracte grammatica voor de While-taal (While\_AST) is te zien in Codevoorbeeld 3-6.
In Codevoorbeeld 3-7 is een instantie van deze grammatica te zien waarin het programma uit Codevoorbeeld 3-1 wat de faculteit berekend is gemodelleerd.

(Codevoorbeeld 3-7)

\[
\begin{align*}
\text{Prog} & \quad \text{= Prog} \ (\text{Opt} \ v) \ (\text{Decl} \ v \ e) \ (\text{Stmt} \ v \ e) \\
\text{Decl} \ v \ e & \quad \text{= Decl} \ v \ v \ (\text{Stmt} \ v \ e) \\
\text{Stmt} \ v \ e & \quad \text{= Comp} \ (\text{Stmt} \ v \ e) \ (\text{Stmt} \ v \ e) \\
\text{Stmt} \ v \ e & \quad \text{= Assign} \ v \ e \\
\text{Var} & \quad \text{= Var} \ \text{String} \\
\text{Expr} & \quad \text{= Expr} \ \text{Expr} \ \text{Opp} \ \text{Expr} \\
\text{Opp} & \quad \text{= Opp} \ \text{String}
\end{align*}
\]

3.2 Concrete syntaxis

Concrete syntaxis of semiabstracte syntaxis bevat naast de benodigde elementen ook elementen die de leesbaarheid van de taal vergroten. Voorbeelden van deze elementen zijn woorden, leestekens en witruimte (white-spaces).

De oorsprong van het vastleggen van concrete syntaxis komt voort uit de types-as-grammatica-aanpak (types-as-grammar approach) die Van Weelden et al gebruikte in [29]. Door het expliciet vastleggen van deze syntaxis kunnen bovenop die definitie functies zoals pretty-printers of parsers gedefinieerd worden.

Een grammatica voor de concrete syntaxis lijkt sterk op een grammatica voor een abstracte syntaxis alleen met meer taalelementen. Al deze extra taalelementen moeten getypeerd worden. Hierdoor wordt de concrete grammatica een stuk groter. De concrete grammatica voor de While-taal (WhileCST) uit Codevoorbeeld 3-6 is te zien in Codevoorbeeld 3-8.
Het eerste deel van de definitie is misschien wat overdreven, alle woorden en tekens die toegevoegd worden ten behoeve van de leesbaarheid moeten van hun eigen type worden voorzien. Hierdoor ontstaan types zoals Als, Dan en Dot.

De types, zoals (StmtC v e), worden complexer door alle extra argumenten die erbij komen ten behoeve van de uiteindelijke tekstuele leesbaarheid. Merk hierbij wel op dat de samenstelling van de argumenten de betekenis representeert en dit niet meer op de dataconstructor neer komt. De twee verschillende interpretaties van het "variabele = 1" voorbeeld uit de introductie van dit hoofdstuk, moeten expliciet worden aangegeven. Dit kan worden gedaan zoals in Codevoorbeeld 3-9, in die grammatica wordt het verschil door de dataconstructoren aangegeven maar ook door de argumenten van de datastructuren.

In Codevoorbeeld 3-10 is wederom het faculiteitprogramma te zien, alleen deze keer in de syntaxis van WhileCST. De concrete syntaxis is vrijwel onleesbaar, maar de CST heeft wel dezelfde bedoelde semantiek als de AST uit Codevoorbeeld 3-7.
(ProgC Programma OpenSquareBracket (Opt (VarC "val"))) CloseSquareBracket NewLine (DeclC Proc 
(Plus Space) (VarC "calcFactorial") (Plus Space) OpenBracket (VarC "x") CloseBracket NewLine 
Begin NewLine (CompC (AssignC (VarC "y") (Plus Space) AssignSign (Plus Space) (ExprC (ExprIC 1)))) Dot NewLine (StmtC (WhileC Zolang (Plus Space) (ExprC (ExprVC (VarC "x"))) (OppC ">")) (ExprC (ExprIC 1))) (Plus Space) Doe NewLine (Block (CompC (AssignC (VarC "y") (Plus Space) 
AssignSign (Plus Space) (ExprC (ExprVC (VarC "y") (OppC "*") (ExprC (VarC "x")))) Dot NewLine (StmtC (AssignC (VarC "x") (Plus Space) AssignSign (Plus Space) (ExprC (ExprVC (VarC "x")))) (OppC "-")) (ExprC (ExprIC 1)))))))) NewLine End) NewLine NewLine Begin NewLine 
(StmtC (AssignC (VarC "result") (Plus Space) AssignSign (Plus Space) (ExprC (CallC VoerUit 
(Plus Space) (VarC "calcFactorial") OpenBracket (ExprC (ExprVC (VarC "val")))))) 
CloseBracket)))) NewLine End)

Codevoorbeeld 3-10
4 De taaltransformator voor taaltransformatie

In de vorige hoofdstukken is achtergrondinformatie en beargumentering gegeven over met welke techniek en methodiek het communicatieprobleem aan te pakken is. In deze sectie wordt het artefact, de taaltransformator, wat het concept voor taalcreatie en transformatie mogelijk maakt beschreven. De taaltransformator maakt het mogelijk om verschillende domeingebieden, die over hetzelfde probleemgebied praten, samen en gelijktijdig tot een softwareoplossing te laten komen. Elk domeingebied gebruikt haar eigen taal/jargon. Deze talen zijn isomorf en kunnen daardoor naar elkaar vertaald worden.

De methodiek achter het concept erkent verschillende niveaus binnen een taal. Namelijk de taal in minimale vorm (abstracte syntaxis) die precies de betekenis representeren en verschillende concrete representatievormen (concrete syntaxis). Een taal heeft één betekenis en kan meerdere representatievormen hebben (views).

De representatie van de betekenis van de taal wordt vastgelegd in een speciale grammatica. Deze grammatica beschrijft de abstracte syntaxis. De abstracte grammatica is minimaal, dat wil zeggen dat zinnen in de taal niets anders bevatten dan betekenisvolle elementen. Aan de taalelementen uit de grammatica kan een semantiek worden gegeven. Zinnen die voldoen aan de abstracte syntaxis vormen een boomstructuur van grammaticaregels. Deze boomstructuur weerspiegelt een uitdrukking in het probleemgebied. Een dergelijke boom wordt de abstracte syntaxboom genoemd, ofwel Abstract Syntax Tree (AST). Echter de AST weerspiegelt alleen een grammaticaal correcte uitdrukking. De semantische analyse en de semantiek die aan de grammatica gegeven is moet vervolgens bepalen of er semantisch iets corrects staat. De AST laat namelijk zinnen toe als “De bal eet een huis.” die grammaticaal correct is, maar semantisch incorrect is.


Door zowel de abstracte als de concrete syntaxis vast te leggen ontstaat een hiërarchie van grammatica's binnen het probleemgebied, zie Figuur 4-1. De CST is ook in een grammatica beschreven en daarom kan de CST zelf ook als AST fungeren als bron voor een nieuwe laag in de hiërarchie.

Door het isomorfisme is het mogelijk om door de hiërarchie te tracereren zonder van betekenis te veranderen, alleen van representatie. Hierdoor kan door enkel een syntaxistransformatie uit te voeren een uitdrukking tussen de verschillende talen overgedragen worden. De uitdrukking behoudt in beide talen dezelfde semantiek. De semantiek van elke uitdrukking in het probleemgebied is gelijk aan bijbehorende de semantiek van de AST.
Het prototype wordt een speciaal soort compiler, deze compiler kan namelijk twee richtingen op: van bron naar doel en van doel terug naar bron. Met een compositie van deze compilers is het mogelijk om door de taalhiërarchie, zoals in Figuur 4-1, te traverseren. Zo kan bijvoorbeeld van CST\(_2\) naar CST\(_1\) getraverseerd worden door de transformaties t5, t3 en t1 te gebruiken. Dit in tegenstelling tot normale compilers, die structuurinformatie na compilatie niet meer opslaan en daardoor niet meer terug kunnen naar de bronstaal.

De architectuur van het prototype is gebaseerd op het M3-model en is geïmplementeerd in functionele programmeertalen. Helaas is de gemaakte transformer gespreid over twee programmeertalen: Clean en Haskell. Het declaratieve gedeelte van het prototype is in beide talen vast te leggen. Er ontstaat echter een probleem bij het definiëren van een executeerbare instantie van de transformatietaal. Dit probleem is type gerelateerd en is te verhelpen door een andere schrijfwijze van types te gebruiken: namelijk als GADTs in plaats van ADTs. Voor technische achtergrond over ADTs en GADTs zie Sectie 2.3.1.1. Om GADTs te kunnen gebruiken is een zijstap gemaakt naar Haskell. Een volledige implementatie van het prototype is mogelijk in Haskell, maar is niet geschreven wegens de beginkeuze van Clean. De vertaling van Clean-code naar Haskell-code zou veel extra simpel werk gekost hebben die nu in het onderzoek gestoken is. In Figuur 4-2 is de technische invulling te zien van de gebruikte architectuur.
5 Taaltransformaties

In dit onderzoek komen twee transformatieklassen voor: een transformatie tussen een boomstructuur en tekst en een transformatie tussen twee boomstructuren. Deze twee transformatieklassen zijn te zien in Figuur 4-1 en worden in dit hoofdstuk beschreven. De verschillende transformatieklassen worden één voor één behandeld, zoals de transformaties genummerd zijn in Figuur 5-1.

Figuur 5-1 - Weergave verschillende transformatieklassen

Eerst zal de transformatie van een boomstructuur naar tekst worden beschreven, deze transformatie wordt pretty-printen genoemd (#1). Dit gebeurt op twee manieren: eerst met de hand en daarna door middel van een generische functie. Na de pretty-print-functie wordt in Sectie 5.2 het parseren van tekst naar een boomstructuur behandeld (#2). Daarna wordt Sectie 5.3 de transformatie van abstracte syntax naar concrete syntax beschreven (#3). Dit gebeurt eerst met een handmatige transformatiefunctie en later wordt naar een transformatietaal toegewerkt door middel van Arrows, deze taal wordt beschreven in Sectie 5.4. In de voorbeelden wordt de abstracte en concrete grammatica van de While-taal gebruikt uit Sectie 3.

5.1 Pretty-printen van grammatica-instanties

Voor de transformatie van een instantie van een boomstructuur naar tekst wordt een zogenaamde pretty-print-functie geschreven. Dat is transformatie #1 in Figuur 5-1. De pretty-print-functie zet een instantie van een grammatica om in tekst, het resultaat van de transformatie is een String. Er wordt uitgegaan van een syntaxis die beschreven is in ADTs omdat de pretty-print-functie voor elke grammatica in ADTs gedefinieerd kan worden, voor zowel abstracte als concrete syntaxis.


Lay-out is belangrijk voor een leesbare en duidelijke uitvoer. De tekstuele representatie van de concrete syntax moet leesbaar en duidelijk zijn. Daarom moet ergens in de combinatie van de pretty-printer en de grammatica witruimte opgenomen worden.

Witruimte is in sommige gevallen contextgevoelig: zo horen bepaalde stukken tekst op hetzelfde inspringniveau te beginnen. Deze contextinformatie moet ook ergens in de pretty-printer of grammatica worden opgenomen, om zo de context in acht te kunnen nemen.

In de komende twee secties wordt eerst de klassengebaseerde aanpak gebruikt voor het definiëren van een pretty-printer. Daarna wordt een generische aanpak gebruikt.
5.1.1 Klassengebaseerde aanpak

Een manier om grammatica-instanties te kunnen printen, in programmeertaalen algemeen, is het schrijven van een functie die alle instanties van een type kan printen en zichzelf recursief aan kan roepen op zijn argumenten indien dat nodig is. Een dergelijke printfunctie kan niet geschreven worden in statisch getypeerde functionele programmeertaalen zoals Haskell en Clean, dit komt omdat de compiler het type voor die functie niet kan afleiden. Een andere mogelijkheid is het definiëren van een printfunctie met een unieke naam voor elk type, maar dat is geen elegante oplossing en zeker niet gebruikersvriendelijk. Het is echter wel mogelijk om een combinatie van de twee eerder genoemde aanpakken te gebruiken door gebruik te maken van klassendeclaratie en overloading. Op deze wijze kan een functie-instantie worden gedefinieerd voor een willekeurig type. Hierdoor ontstaat een familie printfuncties die dezelfde naam hebben.

De klassengebaseerde aanpak is getiteld door de werkwijze die genomen is in de functionele programmeertaal. De printfunctie wordt aan de hand twee voorbeelden beschreven: het eerste voorbeeld beschrijft de declaratie van de functie en het tweede voorbeeld laat zien hoe een klassengebaseerde pretty-print-functie geschreven kan worden.

In Codevoorbeeld 5-1 is een klasse prettyPrint gedefinieerd met als klassenvariabele a. De klassenvariabele is het type van het te printen object. Om contextinformatie door te geven is een record gedefinieerd. In het voorbeeld heet dit record PrintContext. In het record wordt voor de klassengebaseerde aanpak alleen het inspringniveau van de uitvoer bijgehouden.

```haskell
class prettyPrint a :: PrintContext a -> String

Codevoorbeeld 5-1
```

In Codevoorbeeld 5-2 is een printinstantie gegeven voor het type StmtC v e, in dit voorbeeld is gebruik gemaakt van de prettyPrint-klasse. Omdat het type StmtC v e geparameteriseerd is moet in de declaratie aangegeven worden dat de printfunctie ook voor de argumenten gedefinieerd is, in het voorbeeld van het type StmtC v e is dat prettyPrint v en prettyPrint e. Opvallend is dat het vooral simpele String-concatenaties en recursieve aanroepen zijn. Een recursieve aanroep van de printfunctie op een argument levert ook een String op en kan daardoor aan elkaar geconcateneerd worden.

```haskell
instance prettyPrint (StmtC v e) | prettyPrint v & prettyPrint e

where

    prettyPrint ctx (AssignC a b c d e) = ctx.indent
        +++ (prettyPrint emptyCtx a)
        +++ (prettyPrint emptyCtx b)
        +++ (prettyPrint emptyCtx c)
        +++ (prettyPrint emptyCtx d)
        +++ (prettyPrint emptyCtx e)

    prettyPrint ctx (CompC a b c d)  = (prettyPrint ctx a)
        +++ (prettyPrint emptyCtx b)
        +++ (prettyPrint emptyCtx c)
        +++ (prettyPrint ctx d)

    prettyPrint ctx (WhileC a b c d e f g) = (ctx.indent)
        +++ (prettyPrint emptyCtx a)
        +++ (prettyPrint emptyCtx b)
        +++ (prettyPrint emptyCtx c)
        +++ (prettyPrint emptyCtx d)
        +++ (prettyPrint emptyCtx e)
        +++ (prettyPrint emptyCtx f)
        +++ (prettyPrint (indent ctx) g)

instance prettyPrint Als

where

    prettyPrint _ _    = "als"

Codevoorbeeld 5-2
```

Kortom het definiëren prettyPrint-instanties is veel repeterend werk en foutgevoelig door de eenvoud van de functies. Een ander zwaarwegend nadeel van deze aanpak is de grote hoeveelheid simpele code die geproduceerd moet worden. De eenvoud en herhalingsgraad van de functie-instanties lijken beter te passen bij een generische aanpak.
5.1.2 Generische aanpak

Clean biedt de mogelijkheid tot generische functies. Deze functies worden inductief gedefinieerd op de structuur van types. De generische Clean-bibliotheek heeft een generische printfunctie die datatypes kan printen als Clean-syntax. Het is de vraag of, op dezelfde manier als voor de bestaande printfunctie uit generische Clean-bibliotheek, een andere printfunctie gemaakt worden die de syntax in ADTs als seminatuurlijke taal kan printen.

Een generische representatie in Clean bestaat uit UNITs, PAIRS, EITHERs, FIELDs, CONS’, en OBJECTs [2]. Deze datatypes leggen de structuur van types vast. Een generische functie wordt gedefinieerd op de generische representatie en primitieve datatypes. Hierdoor is een generische functie toe te passen op alle types. Indien nodig kunnen uitzonderingen voor types worden gemaakt door deze uitzonderingen expliciet te definiëren. In het definiëren van een uitzondering heeft de generische werkwijze een nadeel en dat is dat de originele typestructuur verlaten wordt. Het verlaten van die structuur heeft tot gevolg dat men de oorspronkelijke types niet meer kan benaderen. Het probleem dat hierdoor ontstaat, komt later aan de orde.

Eerst wordt een voorbeeld van een generische representatie gegeven in Codevoorbeeld 5-3 om duidelijk te maken hoe die eruit ziet. In dit voorbeeld is een lijst als ADT weergegeven in het type ListG a. Deze lijst is daaronder als generische representatie weergegeven in het type ListG a. In hetzelfde voorbeeld is ook de ADT en generische weergave van het type StmtC v e te zien. De laatste generische representatie ziet er complex uit. Het is een voordeel dat de generische representaties niet met de hand gedefinieerd hoeft te worden, ze kunnen door de compiler worden afgeleid.

Nu de generische representatie behandeld is kan begonnen worden aan de definitie van de generische pretty-print-functie. De declaratie van de generische pretty-print-functie lijkt sterk op de klassendeclaratie van de prettyPrint-functie uit Codevoorbeeld 5-1, alleen wordt deze keer het keyword generic in plaats van class gebruikt. De generische printfunctie moet gedefinieerd worden voor de primitieve datatypes zoals Int en String en voor de generische datatypes. De generische pretty-print-instanties voor de primitieve datatypes leveren de waardes van de primitieve datatypes op in de vorm van een String-representatie. Voor de functie-instanties voor de generische datatypes moet de betekenis van het datatype in de context van de printfunctie worden geplaatst. Zo betekent een PAIR dat de twee argumenten achter elkaar horen te staan en een EITHER of het linker of het rechter argument is. De betekenissen van de generische datatypes in de pretty-print-context staan beschreven in Figuur 5-2.

<table>
<thead>
<tr>
<th>ADT in generische wereld</th>
<th>Betekenis in pretty-printer</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIT</td>
<td>Het einde van een tak van de generische representatie.</td>
</tr>
<tr>
<td>PAIR a b</td>
<td>Twee takken aan elkaar plakken.</td>
</tr>
<tr>
<td>EITHER a b</td>
<td>Of de linker of de rechter tak, afhankelijk van de dataconstructor.</td>
</tr>
<tr>
<td>OBJECT a</td>
<td>Objectmarkering, verder geen betekenis.</td>
</tr>
<tr>
<td>CONS a</td>
<td>Constructormarkering, op deze node kan extra informatie van de dataconstructor opgevraagd worden.</td>
</tr>
</tbody>
</table>

Figuur 5-2 - Betekenis generische ADTs in pretty-print-context

De pretty-printer print instanties van grammatica’s. Een grammatica bevat normaalgesproken alleen leesbare elementen. Maar in dit onderzoek zijn spaties, tabs en nieuwe regels ook in de grammatica opgenomen. Deze tekens worden worruimte genoemd. Voor de verschillende grammatica’s, zonder en
met opgenomen witruimte, zijn in dit onderzoek twee generische varianten van de printfunctie geschreven: één waarbij de witruimte niet is mee gemodelleerd in de grammatica maar wordt toegevoegd in de printfunctie en een tweede waarbij witruimte is mee gemodelleerd in de grammatica. In de eerste variant ontstonden problemen bij het achter elkaar zetten van de tekens, namelijk niet tussen alle tekens moet een spație worden gezet. Voor een leesteken hoeft bijvoorbeeld geen spație te gezet te worden. Om dit probleem op te lossen is de tweede variant gemaakt waarbij de witruimte mee gemodelleerd is in de grammatica. Hierdoor wordt de printfunctie een stuk eenvoudiger. De twee generische printfuncties verschillen maar op twee punten: de mee te geven context en de functie-instantie voor het datatype CONS. De oorzaak voor het opnemen van witruimte in de grammatica wordt hieronder beschreven.

Omdat de oorspronkelijke typestructuur verlaten wordt bij het uitvoeren van een generische functie, kan niet meer in de oorspronkelijke types gekeken worden. Bijvoorbeeld een generische functie-instantie toegepast op (Cons 1 Nil) komt uit bij instanties voor de types CONS, EITHER, Int en UNIT, omdat dat de generische instantie daaruit is opgebouwd (zie Codevoorbeeld 5-4). In de generische functie-instanties weet men niet op welke plaats of volgorde de argumenten in de syntaxisboom stonden. Tevens wordt de functie-instantie voor het generische datatype CONS zelfs twee keer aangeroepen: de eerste keer met een RIGHT b als parameter en de tweede keer met een UNIT als parameter en men weet op het moment van definiëren niet over welke CONS het gaat. Daarnaast is ook niet duidelijk met welk oorspronkelijk type gewerkt wordt.

intList  = (Cons 1 Nil)
intListG = OBJECT (CONS (RIGHT (PAIR 1 (LEFT (CONS UNIT))))))

Codevoorbeeld 5-4

Om de originele structuur deels weer terug te krijgen, om de nesting van datatypes weer in handen te krijgen, is de contextvariabele uitgebreid met een lijst met constructornamen. Die lijst bevat de constructornamen van het pad naar de huidige node. Hiervoor moet de generische functie worden aangepast zodat de constructornamen worden toegevoegd aan de context. Op deze wijze kunnen de bovenliggende dataconstructoren uitgelezen worden en kan daar actie op ondernomen worden. Men kan bijvoorbeeld alle elementen uit een lijst scheiden door komma’s, door te controleren of de huidige en de vorige dataconstructor beide Cons zijn. Als dat het geval is kan er een komma worden geprint. Maar voordat specifieke gevallen behandeld worden, wordt eerst de generische printfunctie beschreven.

In Codevoorbeeld 5-5 is de declaratie van de generische printfunctie te zien samen met instanties voor een tweetal primitieve datatypes en de generische datatypes. De gPrettyPrint-instantie voor het type Real is de toString van de Real-waarde en de instantie voor het type String is zijn eigen waarde. De printbetekenis van de generische datatypes is omgetoverd in Clean-code: bij een PAIR worden de argumenten achter elkaar gezet en bij een EITHER wordt een pattern-match gedaan op de dataconstructor waarna de printfunctie voor het specifieke geval uitgevoerd wordt op het argument. De instantie voor het type CONS is het interessantste: hier wordt op basis van de dataconstructor actie ondernomen: de geneste instantie wordt in de context van de huidige dataconstructor geprint. Dataconstructoren kunnen zowel een terminal als non-terminal representeren: terminals moeten worden geprint en non-terminals niet. Om niet voor alle terminals een instantie van de gPrettyPrint-functie te hoeven definiëren, is ervoor gekozen om voor alle dataconstructoren met een ariteit van 0 de constructornamaam op te drukken. Zo wordt het grootste gedeelte van de woorden automatisch geprint, zoals de terminals Als, Dan en Anders. Voor uitzonderingen zoals leestekens en terminals die uit meerdere woorden bestaan moeten wel expliciet functie-instanties gedefinieerd worden. Voorbeelden van deze uitzonderingen staan in Codevoorbeeld 5-6.

Let hierbij op dat CONS tot de generische datatypes behoort en Cons uit Codevoorbeeld 5-4 komt.
\[
GCtx = \{\text{ctors :: [String]}, \text{indent :: String}\}
\]

\[
gPrettyPrint{|\text{Real}|} \_ \_ x = \text{toString} x
\]
\[
gPrettyPrint{|\text{String}|} \_ \_ x = x
\]
\[
gPrettyPrint{|\text{UNIT}|} ctx x = ""
\]
\[
gPrettyPrint{|\text{PAIR}|} fx fy ctx (PAIR x y) = \{fx ctx x\} +++ \{fy ctx y\}
\]
\[
gPrettyPrint{|\text{EITHER}|} fl fr ctx (\text{LEFT} x) = fl ctx x
\]
\[
gPrettyPrint{|\text{EITHER}|} fl fr ctx (\text{RIGHT} x) = fr ctx x
\]
\[
gPrettyPrint{|\text{OBJECT}|} f ctx (\text{FIELD of d}) f ctx (\text{FIELD} x) = f ctx x
\]
\[
gPrettyPrint{|\text{CONS of d}|} f ctx (\text{CONS} x) = \text{indent} +++ \text{prefix} +++ \text{value}
\]

\[
\text{where}
\]
\[
\text{value} = \text{if} (d.gcd_arity > 0) (f ctx` x) \text{curCtor}
\]
\[
\text{curCtor} = d.gcd_name
\]
\[
\text{ctx`} = \{\text{ctx & ctors = \{curCtor : ctx.ctors\}}\}
\]
\[
\text{prefix} = \text{append} ctx`.ctors
\]
\[
\text{indent} = \text{if} (\text{doIdent curCtor indentedCtors}) ctx.indent ""
\]

\[
\text{Codevoorbeeld 5-5}
\]
\[
gPrettyPrint{|\text{Call}|} \_ \_ \_ \_ = \text{"Roep aan"}
\]
\[
gPrettyPrint{|\text{NewLine}|} \_ \_ \_ \_ = \text{"\text{\textbackslash n}"}
\]
\[
gPrettyPrint{|\text{OpenBracket}|} \_ \_ \_ \_ = \text{"\text{\{"}}
\]
\[
gPrettyPrint{|\text{Space}|} \_ \_ \_ \_ = \text{"", "}
\]
\[
\text{Codevoorbeeld 5-6}
\]

Om actie te kunnen ondernemen op basis van de nesting van dataconstructoren zijn de constructornamen van het pad naar de huidige node meegegeven aan de context en is een \text{append}-functie gedefinieerd waarin naar deze lijst kan worden gekeken. In die functie wordt op basis van het pad naar de huidige node gekeken of een extra stukje tekst tussen de woorden moet worden gezet. Deze tekst is het resultaat van de \text{append}-functie. In Codevoorbeeld 5-7 is een instantie van de \text{append}-functie te zien waarin een komma tussen de elementen van een lijst wordt gezet. In die functie-instantie wordt gekeken of de laatste twee dataconstructoren (de eerst twee op de stack) beide "Cons" zijn en als dat het geval is dan wordt er een ", " opgeleverd, in alle andere gevallen wordt een lege String opgeleverd.

\[
\text{append ["Cons", "Cons": xs]} = ", "
\]
\[
\text{append} = ""
\]

\[
\text{Codevoorbeeld 5-7}
\]

Een nadeel van de \text{append}-functie is dat de argumentindex van het geneste type niet bekend is. Neem bijvoorbeeld de datatypes Foo en Bar uit Codevoorbeeld 5-8. Als een spatie moet worden geprint tussen de twee Bar datatypes, dan kan daar een match in de \text{append}-functie voor worden gemaakt. Deze match is dan op een type Bar in een Foo. Alleen kan niet aangegeven worden welke Bar in de Foo bedoeld wordt, beide Bar-types worden geraakt.

:: Foo = Foo Bar Bar
:: Bar = Bar String

\[
\text{append ["Bar", "Foo": xs]} = " "
\]
\[
\text{foo} = \text{Foo (Bar "bar1") (Bar "bar2")}
\]
\[
\text{result} = \text{gPrettyPrint|\text{\^{*}|}} ctx foo // " bar1 bar2"
\]

\[
\text{Codevoorbeeld 5-8}
\]

Bovenstaande code resulteert in een ingewikkelde \text{append}-functie om de verschillende taallelementen netjes weer te geven met de gewenste lay-out. Om geen ingewikkelde \text{append}-functie te hoeven schrijven, is ervoor gekozen om witruimte mee te modelleren in het datatype. Een spatie staat dan gelijk aan een woord qua syntaxiselement. De datatypes worden hierdoor meer opgeblazen en hebben argumenten voor elk teken in de concrete syntaxis. In Codevoorbeeld 5-9 zijn de datatypes Foo en List a geconcretiseerd met extra leestekens en witruimte.

:: FooConcrete = Foo Bar Space Bar
:: ListConcrete a = Nil | Cons a Comma Space (ListConcrete a)

\[
\text{Codevoorbeeld 5-9}
\]
Door het concretiseren van de grammatica moeten extra instanties van de gPrettyPrint-functie gedefinieerd worden namelijk voor de toegevoegde woorden (in dit geval voor Space en Comma). Hierna gaat het printen van het datatype FooConcrete wel zoals gewenst, alleen zal nu een extra komma aan het einde van een niet lege lijst worden geprint bij een ListConcrete a. Dit komt omdat een niet lege lijst altijd van de vorm Cons _ Comma Space Nil is. Het printen van de extra komma aan het einde kan opgelost worden door het herschrijven van de grammatica. Het herschrijven van de grammatica krijgt de voorkeur ten opzichte van het schrijven van een append-functie omdat daar minder geprogrammeerd moet worden en meer gemodelleerd moet worden. De gPrettyPrint-functie wordt hierdoor een stuk eenvoudiger omdat het append-gedeelte niet meer nodig is, zie Codevoorbeeld 5-10.

:: ListConcrete a = Nil | More (ListConcrete` a)
:: ListConcrete` a = Cons a Comma (ListConcrete` a) | Last a

gPrettyPrint(|CONS of d|) f ctx (CONS x) = indent ++ value
  where
    value = if (d.gcd_arity > 0) (f ctx x) d.gcd_name
    indent = if (doIdent d.gcd_name indentedCtors) ctx.indent ""

Codevoorbeeld 5-10

derive gPrettyPrint Als, Dan, Zolang, Doe, Anders, Programma, Begin, End, Proc
derive gPrettyPrint ProgC, DeclC, OppC, VarC, ExprC, ExprC`, StmtC, StmtC`
gPrettyPrint{|VoerUit|} _ _ = "Voer uit"
gPrettyPrint{|NewLine|} _ _ = "\r\n"
gPrettyPrint{|Indent|} _ _ = "\t"
gPrettyPrint{|Quote|} _ _ = ""
gPrettyPrint{|OpenBracket|} _ _ = "(")
gPrettyPrint{|CloseBracket|} _ _ = "")
gPrettyPrint{|Dot|} _ _ = "."
gPrettyPrint{|AssignSign|} _ _ = "="
gPrettyPrint{|Space|} _ _ = " ">
gPrettyPrint{|OpenSquareBracket|} _ _ = "["]
gPrettyPrint{|CloseSquareBracket|} _ _ = "]"
gPrettyPrint(|Block|) f ctx (Block x) = (f {ctx & cindent = ctx.cindent +++ "\t"}) x

Codevoorbeeld 5-11

Door het mee modelleren van de witruimte kan voor de meeste taalelementen uit WhileCST, uit Codevoorbeeld 3-8, de instantie gPrettyPrint worden afgeleid. Alleen voor leestekens, dataconstructoren die meerdere woorden representeren en het Block-element moeten handmatig functies worden gedefinieerd. Door gebruik te maken van de generische gPrettyPrint-functie, uit Codevoorbeeld 5-13, blijft de gPrettyPrint-functie voor WhileCST erg compact. De gehele printfunctie voor WhileCST is te zien in Codevoorbeeld 5-11.

Uit het definiëren van een generische pretty-printer volgt dat pretty-printers eenvoudig te definiëren zijn voor de verschillende grammatica’s. Het grootste deel van de instanties van de pretty-printer kan namelijk worden afgeleid. Hiermee is transformatie #1 uit Figuur 5-1 afgerond.

5.2 Parseren naar grammatica-instanties

Naast de pretty-print-functie uit de vorige sectie is het ook handig om vanuit tekst tot een syntaxisboom te komen. Deze stap is het parseren van tekst. De parseer-functie is #2 in Figuur 5-1. In deze sectie wordt de handmatige en klassengebaseerde aanpak overgeslagen en wordt direct de generische aanpak genomen. De generische representatie is hetzelfde als in Sectie 5.1.2 en de betekenis van de generische datatypes is ook hetzelfde als in Figuur 5-2 alleen deze keer in de context van het parseren in plaats van het printen.

Om de generische parser te maken is gebruik gemaakt van een bestaande set parsercombinatoren [15]. Deze combinatoren zijn gebruikt om de parsers voor de primitieve en generische datatypes te definiëren. Voor de definitie van de parser en de gParse-functie zie Codevoorbeeld 5-12. In de parserdefinitie is de s het type van de invoer en de r het type van de uitvoer. In de gParse-definitie is a het type van het te printen object.

31
Net zoals voor de pretty-printer moeten instanties worden geschreven voor de primitieve en de generische datatypes. Deze instanties zijn te zien in Codevoorbeeld 5-13. Voor een volledig overzicht van parsercombinatoren zie Appendix A.

gParse{|UNIT|} _ = yield UNIT

gParse{|EITHER|} pl pr ctx = ((pl ctx) <@ LEFT) \!/ ((pr ctx) <@ RIGHT)

gParse{|PAIR|} px py ctx = (px ctx) \!/ (py ctx) <@ (PAIR lr)

gParse{|CONS of d|} pc ctx = if (doIdent curCtor indentedCtors)
   ((pWord (fromString ctx.cindent)) \!/ (pc ctx) <@ CONS)
   ((pc ctx) <@ CONS)

where
   curCtor = d.gcd_name
   doIdent x [] = False
   doIdent x [y:ys]
     | x == y = True
     = doIdent x ys

gParse{|OBJECT|} po ctx = (po ctx) <@ OBJECT

gParse{|Int|} _ = pInt

gParse{|Bool|} _ = (pWord ['true'] <@ toString <@ toBool)
     \!/ (pWord ['false'] <@ toString <@ toBool)

gParse{|String|} _ = (!+!) pAlpha <@ toString

gParse{|Real|} _ = pReal

gParse{|Char|} _ = pAlpha

pInt :: (Char -> [(Int,[Char])])
pInt = ((symbol '-' /:
          pNat) \!/ pNat) <@ charList2Int

De parsers voor de generische representatie markeren de geparseerde input met de generische datatypes en het generische raamwerk vertaalt ze naar de gewenste types. De generische parsers voor de primitieve datatypes zijn functies die eenmalig met de hand geschreven zijn. Bijvoorbeeld de parser pInt die eerst een "–" mag parseren en vervolgens één of meer cijfers moet parseren. Het generische raamwerk zorgt ervoor dat de juiste parsers voor de datatypes gecombineerd worden, dit gebeurt op basis van de types.

De generische parsers houden hetzelfde probleem als de parsercombinatoren: ze kunnen niet overweg met linksrecursie. Daarom is in deze scriptie ervoor gekozen om de linksrecursie uit de grammatica te halen. Zodoende kunnen zinnen die voldoen aan die grammatica wel met parsercombinatoren geparseerd worden. In Codevoorbeeld 5-14 zijn de ADTs Stmt` en Expr` te zien, deze zijn aan de grammatica toegevoegd om linksrecursie uit de grammatica te verwijderen. De types met een accent geven de linkerkant van de expressie aan.
datatype opleveren. Deze parsers kunnen eenvoudig gedefinieerd worden met de hulpparser pToken, zie Codevoorbeeld 5-15. De parser pToken herkent het woord dat het eerste argument van de hulpparser is en levert het tweede argument op.

pToken :: String a -> (Parser Char a)
pToken x y = pWord (fromString x) />
yield y

gParse{|Als|} _ = pToken "Als" Als

gParse{|NewLine|} _ = pToken "\n" NewLine

Codevoorbeeld 5-15

Met de linksrecursie uit de grammatica verwijderd en met de parsers voor de verschillende gereserveerde woorden kan voor de grammatica’s (ADTs) de gParse-functie worden afgeleid, zie Codevoorbeeld 5-16. Er moet nog één functie gedefinieerd worden om de gParse-functie precies de uitoefen van de gPrettyPrint-functie te laten parseren. De te definiëren functie bevat de inverse handeling die verricht is voor de pretty-printer voor het datatype Block a zie Codevoorbeeld 5-17.

derive gParse ProgC, DeclC, OppC, VarC, ExprC, ExprC, StmtC, StmtC

Codevoorbeeld 5-16

gParse{|Block|} f ctx = (f {ctx & cindent = ctx.cindent +++ "\t"}) <@ Block

Codevoorbeeld 5-17

In de grammatica is witruimte expliciet gemodelleerd en deze moet precies zo worden geparseerd. Er is geen ruimte voor extra of ontbrekende witruimte. Normaal gesproken zou de parser zo gedefinieerd zijn zodat deze extra witruimte negeert en alleen parseert wat nodig is. Dit doet de afgeleide gParse-functie niet. Hiervoor moet een handmatige handeling worden gedaan om de generische parser extra witruimte te laten negeren. Om witruimte te negeren is een parser gedefinieerd genaamd pSkipSpace. Deze parser kan voor of achter een ander parser worden gezet om de witruimte te negeren. Op dit punt begint het generische raamwerk wat te knellen: er kan geen gebruik worden gemaakt van de afgeleide generische functies. Dit komt omdat de functie handmatig overschreven wordt om de parser pSkipSpace toe te voegen. De afgeleide functie is dan verdwenen en de oorspronkelijke functionaliteit is niet meer aan te roepen. In het generische raamwerk zijn geen verwijzingen naar de overschreven functionaliteit, zoals in imperatieve programmeertalen een verwijzing is naar parent of base. De ingreep die nodig is om extra witruimte te negeren zorgt ervoor dat de hele functie-instantie met de hand gedefinieerd moet worden, dus ook het deel wat anders afgeleid wordt, zie Codevoorbeeld 5-18.

pSkipSpace :: (Parser Char a) -> Parser Char a
pSkipSpace p = ((!*!) (pWord [' '] \!/ pWord ['\t']) ) />

Codevoorbeeld 5-18

Een aandachtspunt bij het negeren van witruimte is dat de generische parser wel de datatypes moet parseren die gedefinieerd zijn, dus ook meegemodelleerde witruimte. Dus wanneer de generische parsers overschreven worden met handgemaakte instanties, die extra witruimte negeren of witruimte introduceren in het datatype, moet wel rekening gehouden worden dat ze de benodigde witruimte wel parseren. Dit betekent meer handwerk en meer code om alle uitzonderingen te definiëren. Het probleem van extra of ontbrekende witruimte is ook op te lossen binnen de methode van dit onderzoek. Er moet dan een grammatica gemaakt worden die speciaal gemaakt is om tekst te parseren waarin witruimte niet is opgenomen, zie Codevoorbeeld 5-19. De parser kan dan alle witruimte negeren. Hiermee wordt de taalhierarchie vergroot en is een extra transformatie nodig tussen de AST en de grammatica voor het parseren, zie Figuur 5-3.
:: StmtC\(_{\text{print}}\) v e = AssignC v Space AssignSign Space e
| IfC Als Space e Spaces Dan NewLine (StmtBlock v e) NewLine
  | Anders NewLine (StmtBlock v e)
| WhileC Zolang Space e Spaces Doe NewLine (StmtBlock v e)
| SkipC

:: StmtC\(_{\text{parse}}\) v e = AssignC v AssignSign e
| IfC Als e Spaces Dan (StmtBlock v e) Anders (StmtBlock v e)
| WhileC Zolang e Doe (StmtBlock v e)
| SkipC

**Codevoorbeeld 5-19**

```
AST ←→ tekst ↔ pretty printing / parsing
        ↔ transformatie

CST\(_{\text{print}}\) ← tekst

CST\(_{\text{parse}}\) ← tekst
```

**Figuur 5-3 - AST met aparte grammatica’s voor te printen of te parseren**

```haskell
eq :: (ProgC VarC ExprC) → Bool
eq input = gEq{/\}$ input parsed
where
  [(parsed, _):_] = gParse{/\}$ input
  printed = fromString (gPrettyPrint{/\}$ ctx input)
```

**Codevoorbeeld 5-20**

De generische parseerfunctie uit dit onderzoek is de inverse van de generische printfunctie. Daarnaast kan de parseerfunctie overweg met extra witrui mte rondom statements en expressies. Om te controleren of de parseerfunctie echt de inverse is moet de parser worden toegepast op de uitvoer van de printer. De uitvoer van de parser moet dan gelijk zijn aan de invoer die aan de pretty-printer gegeven is. Deze regel is weergegeven in Codevoorbeeld 5-20, de invoer in dat voorbeeld is een AST van het type ProgC VarC ExprC.

Door het maken van een generische parser kan een groot deel van de parser voor een grammatica worden afgeleid. Voor de terminals moeten echter nog handmatig instanties worden gedefinieerd. Hiermee is transformatie #2 uit Figuur 5-1 afgerond en is de transformatie in beide richtingen tussen tekst en syntaxisboom beschreven.

### 5.3 Syntaxistransformaties

In de vorige secties zijn twee transformaties getoond waarmee tussen ADTs en tekst kan worden gewisseld. Dat waren transformatie #1 en #2 in Figuur 5-1. Voor beide transformaties is de abstracte grammatica uitgebreid met extra argumenten. Zo ontstaat een concrete grammatica ofwel een semiabstracte grammatica en dus een scheiding tussen de abstracte en de concrete syntax. Dit resulteert in dat de presentatie van de taal los blijft van de essentie. Hierdoor zijn meerdere representaties mogelijk. Tussen de verschillende representaties kunnen transformaties worden gedefinieerd om kennis in een andere representatie te presenteren.

In deze sectie worden een tweetal methodes getoond om een transformatie te definiëren tussen twee syntaxisbomen. Deze transformatie is in Figuur 5-1 transformatie #3 tussen de AST en CST. In dat figuur staat een dubbele pijl tussen AST en CST maar de transformatie wordt één richting op beschreven. Beide richtingen zijn te specificeren. Daarnaast maakt het voor de transformatie niet uit of de transformatie gedefinieerd is tussen ASTs of CSTs. In Sectie 5.3.1 is een handmatige versie beschreven en in Sectie 5.3.2 wordt een Arrow-notatie gebruikt.
5.3.1 Handmatige grammaticatransformatie

Eerst wordt een handmatige transformatie tussen twee grammatica’s beschreven. Er wordt, net zoals bij de `prettyPrint`-functie, een klassembleeratie gemaakt zodat voor elk type een instantie met dezelfde naam kan worden gedefinieerd. Deze keer is zowel het bron type als het doeltype ge parameteriseerd, zie Codevoorbeeld 5-21. In dat voorbeeld is `a` het type van de invoer en `b` het type van de uitvoer. Voor ge parameteriseerde type-instanties moet wederom worden verplicht dat de functie ook gedefinieerd is voor de parameters.

```haskell
class trans a b :: a -> b
instance trans (Statement v e) (StatementC vC eC) | trans v vC & trans e eC
instance trans Var VarC
instance trans Expr ExprC
Codevoorbeeld 5-21
```

Voor een transformatie tussen twee grammatica’s moet voor elk datatype in de bron grammatica een instantie van de klasse `trans` worden gedefinieerd. In de `While`-taal worden ge parameteriseerde datatypes gebruikt en daarom moet bij de ge parameteriseerde datatypes vereist worden dat de `trans`-functie ook voor de parameters is gedefinieerd of de parameters moeten expliciet gedefinieerd zijn in de transformatiefunctie.

In de handmatige versie van de transformatie wordt op basis van het patroon van het datatype getransformeerd. Op deze wijze krijgt men de juiste argumenten in handen en kan op basis daarvan een instantie van het gewenste doeltype gecreëerd worden, zie Codevoorbeeld 5-22.

```haskell
//trans :: (Statement v e) -> (StatementC vC eC)
instance trans (Statement v e) (StatementC vC eC) | trans v vC & trans e eC
where
  trans (Assign v e) = AssignC (trans v) AssignSign (trans e)
  trans (Skip) = SkipC
  trans (Comp s1 s2) = CompC (trans s1) Dot NewLine (trans s2)
  trans (If e s1 s2) = IfC Als (trans e) Dan NewLine Indent (trans s1) Anders NewLine Indent (trans s2)
  trans (While e s) = WhileC Zolang (trans e) Doe NewLine Indent (trans s)
Codevoorbeeld 5-22
```

Wanneer voor elk datatype van de grammatica een transformatie-instantie geschreven wordt, zijn instanties van de bron taal transformeerbaar naar de doel taal. Transformatie #3 uit Figuur 5-1 is hiermee één richting op beschreven en het doel van de transformatie is hiermee bereikt. De transformatie is echter hard geprogrammeerd in Clean en hierdoor kan de transformatie alleen uitgevoerd worden en is verder niets met de transformatiedefinitie mogelijk.

5.3.2 Arrow-gebaseerde grammaticatransformatie

Om de transformatie stapsgewijs te beschrijven wordt de creatie van het taalelement uit doel taal opengebroken en worden de argumenten van het doelelement één voor één in het element geplaatst. Zodoende ontstaan stappen die één voor één uitgevoerd kunnen worden en later samen kunnen worden gevoegd tot het doelelement. Het samenvoegen van de transformatiestappen gebeurt op basis van het combineren van Arrows [9].

Voordat overgegaan wordt tot het gebruik van een Arrow-notatie wordt eerst de aanleiding tot het gebruik van Arrows duidelijk gemaakt. Zoals eerder vermeld hebben types één of meerdere data constructoren. Deze data constructoren zijn functies die met de juiste argumenten een instantie van het type de data constructor opleveren, zie Sectie 2.3.1. Dus datatypes kunnen worden opgebouwd door middel van functiecompositie. Daarnaast zijn types rechtssassociatief, dat wil zeggen dat de argumenten één voor één meegegeven kunnen worden, de argumenten kunnen één voor één meegegeven worden totdat het resultaat `kind *` wordt. Neem bijvoorbeeld de data constructor `AssignC` van het type `StmtC a b`. Na het meegeven van één argument verandert het type, zie Codevoorbeeld 5-23. De variabele `assignC` heeft één argument minder nodig dan de variabele `assignC` om `kind *` te worden. In dit voorbeeld is te zien dat de dan nog ongebonden variabele `a` door het meegeven van een argument getypeerd wordt met het type van het argument, in dit geval is dat het type `Var`. 

35
assignC :: a -> (Plus Space) -> AssignSign -> (Plus Space) -> b -> StmtC a b
assignC = AssignC
assignC' :: (Plus Space) -> AssignSign -> (Plus Space) -> b -> StmtC Var b
assignC' = assignC (Var "x")
assignC'' :: AssignSign -> (Plus Space) -> b -> StmtC Var b
assignC'' = assignC' AssignSign
assignC''' :: b -> StmtC Var b
assignC''' = assignC'' (Single Space)
assignC'''' :: (Plus Space) -> b -> StmtC Var b
assignC'''' = assignC''' AssignSign
assignC''''' :: (Plus Space) -> (AssignSign -> (Plus Space) -> (b -> (StmtC a b)))
assignC''''' = assignC'''' AssignSign
assignC'''''' :: StmtC Var Expr
assignC'''''' = assignC''''' (ExprC (ExprVC (VarC "Var")) (OppC "+")) (ExprIC 1))

Codevoorbeeld 5-23

Met de associativiteit van functies en het gewenste doeltype is precies bekend in welke volgorde argumenten aan de dataconstructor meegegeven moeten worden om een het doeltype te krijgen. Elk meegegeven argument levert een type op wat precies één argument minder nodig heeft om kind * te worden. Het concrete datatype AssignC heeft vijf argumenten, dit resulteert in vijf functiestappen totdat het een instantie van kind * oplevert.

In Codevoorbeeld 5-24 zijn alle vijf de functies (f, g, h, i en j) voor de dataconstructor AssignC volledig uitgeschreven inclusief typedefinities. Hieraan valt op dat telkens het tweede argument toegepast wordt op het eerste argument. Sterker nog, de compiler kan zelfs de complexe types afleiden als de typedefinities weggelaten worden. Nog sterker: de functies kunnen zelfs gegenereerd worden tot één enkele functie. Deze functie krijgt dan het type a -> (a -> b) -> b, zie Codevoorbeeld 5-25.

Gebruikmakend van die functie, kunnen de vijf stappen (f", g", h", i" en j") voor de dataconstructor AssignC beschreven worden. Het enige wat nog moet gebeuren is het in de correcte volgorde plaatsen van de functies.

f :: a (a -> ((Plus Space) -> (AssignSign -> ((Plus Space) -> (b -> (StmtC a b))))) -> ((Plus Space) -> (AssignSign -> ((Plus Space) -> (b -> (StmtC a b)))))
f x y = y x

g :: (Plus Space) ((Plus Space) -> (AssignSign -> ((Plus Space) -> (b -> (StmtC a b))))) -> (AssignSign -> ((Plus Space) -> (b -> (StmtC a b)))
g x y = y x

h :: AssignSign (AssignSign -> ((Plus Space) -> (b -> (StmtC a b))))) -> ((Plus Space) -> (b -> (StmtC a b)))
h x y = y x

i :: (Plus Space) ((Plus Space) -> (b -> (StmtC a b))) -> (b -> (StmtC a b))
i x y = y x

j :: b (b -> (StmtC a b)) -> (StmtC a b)
j x y = y x

f' = f (VarC "Var")
g' = g (Plus Space)
h' = h AssignSign
i' = i (Plus Space)
j' = j (ExprC (ExprVC (VarC "Var")) (OppC "+")) (ExprIC 1))

assign :: StmtC VarC ExprC
assign = (j' o i' o h' o g' o f') AssignC

Codevoorbeeld 5-24

De gegeneraliseerde code uit Codevoorbeeld 5-25 is een stuk compacter dan de oorspronkelijke code; of de leesbaarheid van de code duidelijker geworden is tot daar aan toe. Waar het om gaat is dat door middel van een reeks van functiecomposities het doeltype verkregen wordt. Met de reeks functiecomposities komen we bij de Arrow-notatie aan.

```haskell
class Arrow arr
where
    arr :: (a -> b) -> arr a b
    (>>>) :: (arr a b) (arr b c) -> arr a c

Codevoorbeeld 5-26
```

```haskell
fArr :: a (VarC -> b) b | Arrow a
fArr = arr f`

gArr :: a ((Plus Space) -> b) b | Arrow a
gArr = arr g`

Codevoorbeeld 5-26
```

```haskell
hArr :: a (AssignSign -> b) b | Arrow a
hArr = arr h`
iArr :: a ((Plus Space) -> b) b | Arrow a
iArr = arr i`

Codevoorbeeld 5-26
```

```haskell
jArr :: a (ExprC -> b) b | Arrow a
jArr = arr j`

Codevoorbeeld 5-26
```

```haskell
trans`` = (fArr >>> gArr >>> hArr >>> iArr >>> jArr)
assign`` = trans`` AssignC

Codevoorbeeld 5-27
```

De applicatie van de verschillende functies wordt nu door de `>>>`-combinator afgehandeld. Door de compositie van de Arrows ontstaat een nieuwe functie die gegeven de dataconstructor `AssignC` een correcte instantie van het type `StmtC VarC ExprC` oplevert.

Als goed naar de types gekeken wordt die de compiler heeft afgeleid (in Codevoorbeeld 5-27), dan is te zien dat `gArr` en `iArr` hetzelfde type hebben. Dit zorgt ervoor dat de Arrows met hetzelfde type elkaar kunnen vervangen. Dus Arrows kunnen her gebruikt worden op verschillende posities in de compositie en zelfs in andere composities, zie Codevoorbeeld 5-28. Daarin worden `arrVarC` en `arrSpaces` meerder malen gebruikt.

```haskell
arrVarC = arrVarC >>> arrSpaces >>> arrAssignSign >>> arrSpaces
arrCall = arrCallC

Codevoorbeeld 5-28
```

In bovenstaande voorbeelden zijn de in te vullen argumenten met de hand vastgelegd. Als de creatie van de argumenten afhankelijk van de invoer gemaakt kan worden, dan kan door middel van Arrows de invoer naar het gewenste doeltype getransformeerd worden. Een transformatie gaat vanuit een bronotype naar een doeltype. De functie krijgt een instantie van het bronotype als argument en moet daar een instantie van het doeltype van maken. Om alle Arrows van voldoende informatie te voorzien wordt het argument meegeven aan de Arrows van de transformatie. Zo kan in de Arrow op basis van het argument actie ondernomen worden. Niet alle Arrows creëren
elementen op basis van de invoer, zo zijn er transformatiestappen die nieuwe argumenten introderen. Deze transformatiestappen doen dan niets met de meegegeven invoer. De transformatie van het abstracte taalelement `Assign v e` naar het concrete taalelement `AssignC v` bestaat uit vijf stappen. Twee van die vijf stappen zijn deeltransformaties die een argument uit het abstracte type nodig hebben, dit zijn de deeltransformaties voor de argumenten `v` en `e`. In Codevoorbeeld 5-29 is de transformatie met behulp van Arrows te zien, van `Stmt` a b naar `StmtC` a b. Daarin zijn de twee deeltransformaties te zien die een argument uit de bron gebruiken, de andere drie deeltransformaties introduceren datatypes zonder gebruik te maken van de invoer.

```haskell
assignTrans :: (Stmt` a b) -> StmtC` a b
assignTrans x = ((functions (Assign v e) -> \f -> f (varTrans v)) x
>>>(functions (\_\_\_\_\_f -> f (Plus Space)) x)
>>>(functions (\_\_\_\_\_f -> f AssignSign) x)
>>>(functions (\_\_\_\_\_f -> f (Plus Space)) x)
>>>(functions (Assign v e) -> \f -> f (exprTrans e)) x
) AssignC
```

**Codevoorbeeld 5-29**

Op de wijze uit Codevoorbeeld 5-29 kan voor elk taalelement uit de abstracte taal een transformatieregel gedefinieerd worden. Echter het schijven van de Arrow-transformatie is een tamelijk technische bezigheid. Het patroon van het argument moet namelijk worden vergeleken met de juiste dataconstructor en de juiste argumenten moeten uit de invoer gefilterd worden. Daarnaast moet de gehele transformatie in een klassendeclaratie worden gestopt om dezelfde functienaam te kunnen hanteren. In de instantie van de klassendeclaratie moet een extra controle op de dataconstructor worden gedaan bij datatypes met meerdere dataconstructoren. Dit komt omdat de Cleancompiler de afgeleide instantie niet doortrekt naar de linkerkaart van de expressie (LHS), zie Codevoorbeeld 5-30. In dit codevoorbeeld moet op de variabele `x` een extra pattern-match worden gedaan. Een volledige transformatiespecificatie tussen `Prog Var Expr` en `ProgC VarC ExprC` is te vinden in Appendix B.

```haskell
arrSpaces = arr (functions (\f -> f (Plus Space)))
instance Transform (Stmt v e) (StmtC vc ec) | Transform v vc & Transform e ec
where
    Transform x=:(Assign _ _) = (arr (functions (Assign v e) -> \f -> f (varTrans v)) x
>>> arrSpaces
>>> arrAssignSign
>>> arrSpaces
>>> arr (functions (Assign v e) -> \f -> f (exprTrans e)) x
) AssignC
```

**Codevoorbeeld 5-30**

Door gebruik te maken van Arrows zijn de transformatiestappen één voor één te definiëren en later te combineren door middel van de Arrow-combinators. Hiermee is wederom het doel van transformatie #3 uit Figuur 5-1 bereikt, alleen is deze keer de transformatie opgebouwd uit elementaire transformatiedelen. Echter de transformatiedelen zijn alsnog uitgeprogrammeerd in de programmentaal, waardoor de specificatie alleen uitvoerbaar is. Met gebruikte methodiek van dit onderzoek kunnen talen beschreven worden in datatypes, tevens kunnen deze datatypes (ofwel grammatica's) worden getransformeerd naar andere datatypes. Op deze wijze kan een grammatica worden gedefinieerd om de Arrow-gebaseerde transformaties in te beschrijven zodat die transformaties geparseerd, geprint en uitgevoerd kunnen worden.

**5.4 Een taal voor taaltransformaties**

In de vorige sectie is te lezen dat met behulp van Arrows een stapsgewijze transformatiefunctie te maken is. Alleen de transformaties zijn niet eenvoudig te schrijven, noch te lezen. In deze sectie wordt transformatie #3 uit Figuur 5-1 wederom beschreven met het doel om een transformatiefunctie te krijgen met de stapsgewijze definitie uit Sectie 5.3.2 die gemodelleerd is in een grammatica. Zodoende kan een transformatie die gedefinieerd is in een grammatica, anders gerepresenteerd worden door de methodiek die in dit onderzoek wordt gebruikt.
In deze sectie wordt eerst een abstractie gegeven van de Arrow-gebaseerde transformatie. Daarna wordt een pretty-printer getoond voor deze transformatietaal. De sectie eindigt met poging tot het definiëren van een parser voor de transformatietaal.

Wanneer naar de Arrow-gebaseerde transformatie-instanties gekeken wordt (zie Appendix B), dan is te zien dat de transformatie uit drie delen bestaat:
1. Een transformatiedefinitie met een bronotype en doeltype,
2. Een transformatieregel voor elk van de dataconstructoren van de brontypes,

Het belangrijkste deel van de transformatie is de Arrow, daarmee wordt een functie geconstrueerd die een correcte instantie van het doeltype oplevert. Wanneer van de Arrow-gebaseerde transformatie-instanties geabstraheerd wordt, komen er zes verschillende acties (ofwel taalelementen) voor:
1. Het introduceren van een nieuw argument,
2. Het transformeren van een argument uit het bronotype,
3. Het aanpassen van een resultaat wat opgeleverd is door een transformatiestap van een argument uit het bronotype,
4. Het plaatsen van een dataconstructor om een functie af te sluiten,
5. Het plaatsen van een dataconstructor zonder transformatiestappen,
6. Het toepassen van twee Arrows op elkaar.

Wanneer deze punten als een grammatica worden beschreven, moet ook een functie gemaakt worden die een uitdrukking in de taal uit kan voeren. Helaas biedt Clean niet de mogelijkheid om een combinatie van ADTs en een interpreterfunctie te maken die typecorrect is. Een dergelijke combinatie kan wel gemaakt worden met GADTs, maar GADTs zijn helaas nog niet mogelijk in Clean. Daarom wordt voor deze aanpak een zijstap naar Haskell gemaakt.

De drie transformatiedelen vertalen zich naar drie GADTs en binnen de drie GADTs vertalen de zes transformatieacties zich naar dataconstructoren. Om de transformatieregel aan het bijbehorende taalelement te koppelen zijn twee extra dataconstructoren gebruikt. De eerste voor het koppelen van het taalelement aan de transformatieregel (TransRule) en de tweede voor het afsluiten van de lijst met mogelijke transformatieregels (EndRule). De drie GADTs zijn te zien in Codevoorbeeld 5-31 en de betekenis van de argumenten van de GADTs is te vinden in Figuur 5-4. Bij dit figuur moet wel worden gezegd dat de typering van rigid variabelen niet voor het hele type geldt maar alleen voor de huidige dataconstructor. In de specificatie is de benaming van de types met hetzelfde type en doel gelijk gehouden.

data TransLang a b where
    EndRule  :: TransLang a b
    TransRule :: (b -> Bool, TransRule c a b) -> TransLang a b -> TransLang a b

data TransRule a b c where
    PutCtor  :: ArrLang a b c -> a -> TransRule a b c
    PutCtorE :: b -> TransRule a b c

data ArrLang a b c where
    TransIntr :: (c -> a -> b) -> ArrLang a b c
    TransArrow :: ArrLang a d c -> ArrLang d b c -> ArrLang a b c
    TransArg :: (c -> e) -> TransLang f e -> ArrLang (f -> b) c
    TransArgP :: (c -> e) -> TransLang f e -> (f -> g) -> ArrLang (g -> b) c

Codevoorbeeld 5-31

---

6 Een rigid variabele is niet gebonden door zijn type, het type is alleen bekend binnen de huidige instantie en wordt bepaald door type-inferentie.
Naam typeparameter | Betekenis
---|---
**TransLang**
|  
a | Het type van de uitvoer van de transformatie  
b | Het type van de invoer van de transformatie  
c | Rigid, wordt bepaald door TransRule => het type van de invoer  
**TransRule**
|  
a | Het type van de te plaatsen dataconstructor  
b | Het type van de uitvoer  
c | Het type van de invoer  
**ArrLang**
|  
a | Het type van de te plaatsen dataconstructor  
b | Het type van de uitvoer  
c | Het type van de invoer  
d | Rigid, het tijdelijke type dat over wordt gedragen tussen de twee Arrows  
e | Rigid, het type van het te transformeren argument uit de invoer  
f | Rigid, het type van de uitvoer na transformatie argument  
g | Rigid, het type van de aangepaste uitvoer na transformatie argument

Figuur 5-4 - Betekenis typeparameters van de Arrow-taal GADTs

De Arrow-gebaseerde transformatie wordt uitgevoerd op basis van functieapplicatie en is gedefinieerd in GADTs. Deze GADTs kunnen alleen de transformatie beschrijven maar niet uitvoeren. Dit komt doordat de transformatie declaratief is en grammatica beschreven. Een instantie van de transformatiegrammatica kan een correcte transformatiebeschrijving tussen twee grammatica’s bevatten. Om de transformatiebeschrijving uit te voeren is een evaluatiefunctie nodig die het model weer omzet in een uitvoerbare Arrow-variant, zie Codevoorbeeld 5-32.

```haskell
transEval :: TransLang a b -> b -> a
transEval (TransRule (p, t) xs) i = if p i then (ruleEval t i)
    else transEval xs i

ruleEval :: (TransRule a b c) -> c -> b
ruleEval (PutCtor 1 c) a = (arrEval l a) c
ruleEval (PutCtorE c) _ = c

arrEval :: ArrLang a b c -> c -> (a -> b)
arrEval (TransIntr f) i = arr (f i)
arrEval (TransArrow f g) i = (arrEval f i) >>> (arrEval g i)
arrEval (TransArg f l) i = arr (\x -> x (transEval l (f i)))
arrEval (TransArgP f l g) i = arr (\x -> x (g (transEval l (f i))))
```

Codevoorbeeld 5-32

Buiten de transformatietaal en de evaluatiefuncties zijn per dataconstructor in de brontaal een aantal functies nodig om een transformatie te kunnen definieren, namelijk een functie die de dataconstructor controleert en functies die een argument van het datatype op kunnen leveren. De controlefunctie is een predicaat dat controleert of de instantie een bepaalde dataconstructor heeft. Deze dataconstructor moet namelijk overeenkomen met de dataconstructor van de transformatieregel. Daarnaast moet voor elk van de benodigde argumenten die tijdens de transformatie gebruikt worden een functie beschikbaar zijn die het argument uit de broninstantie haalt. Beide genoemde functies zijn eenvoudige functies, zie Codevoorbeeld 5-33. De predicaaten zijn aangeduid met een prefix p gevolgd door de naam van de dataconstructor en de functies die de argumenten op te leveren hebben de naam van de dataconstructor gevolgd door de index van het op te leveren argument.

```haskell
:: Expr   = Expr Expr` Opp Expr  
| Expr` Expr`

pExpr (Expr _ _ _)  = True
pExpr  = False
pExpr` (Expr` _)  = True
pExpr` = False
expr1 (Expr a b c) = a
expr2 (Expr a b c) = b
expr3 (Expr a b c) = c
expr`1 (Expr` a) = a
```

Codevoorbeeld 5-33
Met de transformatietaal, evaluatiefuncties, predicaten en de functies die argumenten opleveren kunnen transformaties gedeclareerd en uitgevoerd worden. In Codevoorbeeld 5-34 is een voorbeeld gegeven van een gedeelte van de transformatiedeclaratie van het type Stmt’ naar StmtC’, dit is een deel van de transformatie van WhileAST naar WhileCST.

\[
\text{stmt’2StmtC’} = \text{TransRule (pAssign, assign2AssignCRule)} \\
\quad (\text{TransRule (pIf, if2IfCRule)} \\
\quad (\text{TransRule (pWhile, while2WhileCRule)} \\
\quad (\text{TransRule (pSkip, skip2SkipCRule)} \\
\quad \text{EndRule})))
\]

\[
\text{while2WhileCRule} = \text{PutCtor while2WhileCArrow WhileC}
\]

\[
\text{while2WhileCArrow} = \text{TransArrow (TransIntr (\_ -> \_ -> f Zolang))} \\
\quad (\text{TransArrow (TransIntr (\_ -> \_ -> f Space))} \\
\quad (\text{TransArrow (TransIntr (\_ -> \_ -> f expr2ExprC)} \\
\quad (\text{TransArrow (TransIntr (\_ -> \_ -> f Space))} \\
\quad (\text{TransArrow (TransIntr (\_ -> \_ -> f Doe)} \\
\quad (\text{TransArrow (TransIntr (\_ -> \_ -> f NewLine)} \\
\quad (\text{TransArgP (\(\text{While a b}ighthook\rightarrow\text{b})\text{ stmt2StmtC Block}})))))))))
\]

Codevoorbeeld 5-34

Door voor alle taallelementen uit WhileAST transformatieregels te definiëren, zoals in Codevoorbeeld 5-34 gedaan is voor het type Stmt’, kunnen abstracte expressies naar concrete expressies getransformeerd worden, zie Codevoorbeeld 5-35. In dit codevoorbeeld is te zien dat instanties getransformeerd kunnen worden.

\[
\text{stmt} = \text{Stmt (While (Expr (ExprV (Var "x")) (Opp "\geq") (Expr’ (ExprI 1))))} \\
\quad (\text{Stmt (Assign (Var "x") (Expr (ExprV (Var "x")) (Opp "\"-") (Expr’ (ExprI 1))))})
\]

\[
\text{stmtC} = \text{transEval stmt2StmtC stmt}
\]

\[
\text{stmtC} = \text{StmtC (WhileC Zolang Space (ExprC (ExprVC (VarC "x")) (OppC "\geq") (Expr’ (ExprIC 1))))} \\
\quad (\text{StmtC (AssignC (VarC "x") Space AssignSign Space (ExprC (ExprVC (VarC "x") (OppC "\"-") (Expr’ (ExprIC 1)))))})
\]

Codevoorbeeld 5-35

Met het definiëren van een grammatica voor de Arrow-gebaseerde transformatie is een scheiding aangebracht tussen de definitie van een transformatie en de executie daarvan. Een gevolg hiervan is dat de definitie gevangen is in een grammatica en daar functies op te definiëren zijn zoals een pretty-printer of een parser.

In Sectie 5.3.2 is gezegd dat de Arrow-gebaseerde transformatiedefinitie moeilijk leesbaar is, maar de transformatie-instantie uit Codevoorbeeld 5-34 is niet duidelijker. De definitie ziet er nog steeds technisch uit en is zelfs complexer geworden door de extra datatypes die erbij zijn gekomen. Echter het verschil tussen de taal uit Sectie 5.3.2 en de taal uit deze sectie is dat in deze variant de declaratie en evaluatie gescheiden zijn. De declaratie van de transformatie zit nu in een GADT en transformatiespecificaties (GADT-instanties) kunnen door een evaluatiefunctie uitgevoerd worden. In de volgende sectie wordt een pretty-printer voor de transformatietaal beschreven, met de techniek uit Sectie 5.1. Dit levert een leesbare tekstuele variant van de transformatietaal op.

### 5.4.1 Een pretty-printer voor de transformatietaal

De transformatietaal is beschreven met GADTs in Haskell en in deze sectie wordt een pretty-printer beschreven voor die transformatietaal om een leesbare variant van de transformatie te krijgen. Echter de pretty-printer uit Sectie 5.1 is beschreven in ADTs in Clean. GADTs zijn niet letterlijk over te zetten naar ADTs. Daarom wordt de pretty-printer in Haskell geschreven. Een verschil met het maken van een pretty-printer voor de Arrow-taal, ten opzichte van de gemaakte pretty-printers in Sectie 5.1, is dat de Arrow-taal argumenten heeft van hogere orde. Zo heeft de dataconstructor TransRule een argument \(b \leftrightarrow \text{Bool}\), \(\text{wat kind * } \rightarrow *\) is en de dataconstructor TransIntr heeft zelfs een parameter met het type \(c \rightarrow a \rightarrow b \text{ wat kind * } \rightarrow * \rightarrow *\). De vraag is hoe kunnen deze functies worden geprint?

Het printen van functies kan op verschillende manieren, bijvoorbeeld de creatie van functies kan in de taal opgenomen worden en daar kan naar worden verwezen of er kan een koppeling gemaakt worden.
tussen gereserveerde woorden en de benodigde functies. Om de transformatietaal simpel te houden is ervoor gekozen om een koppeling te maken tussen woorden en de daadwerkelijke functies. Dit is gedaan door functies te labelen. De labels worden geprint en gerelateerd aan de bijbehorende functie. Echter bieden de gebruikte generische raamwerken van zowel Haskell als Clean geen mogelijkheid voor het afleiden van instanties met functietypes. Daarom is ervoor gekozen om eenvoudige `toString`-functies te schrijven voor de GADTs.

Om de Arrow-taal te kunnen printen is een concrete grammatica gemaakt, waarin de hogere orde argumenten zijn gedecoreerd met een label, zie Codevoorbeeld 5-36. Het label doet niets anders dan een extra `String`-waarde aan het type toevoegen, deze waarde is de koppeling tussen de daadwerkelijke functie en de representatie als `String`.

\[
:: \text{Label} \ a = \text{Label} \ \text{String} \ a
\]

\[
data \ \text{TransLangC} \ a \ b \ \text{where}
\]

\[
\text{EndRuleC} :: \text{TransLangC} \ a \ b
\]
\[
\text{TransRuleC} :: ((\text{Label} \ (b \rightarrow \text{Bool})), \text{TransRuleC} \ c \ a \ b) \rightarrow \text{TransLangC} \ a \ b \rightarrow \text{TransLangC} \ a \ b
\]

\[
data \ \text{TransRuleC} \ a \ b \ c \ \text{where}
\]

\[
\text{PutCtorC} :: \text{ArrLangC} \ a \ b \ c \rightarrow (\text{Label} \ a) \rightarrow \text{TransRuleC} \ a \ b \ c
\]
\[
\text{PutCtorEC} :: (\text{Label} \ b) \rightarrow \text{TransRuleC} \ a \ b \ c
\]

\[
data \ \text{ArrLangC} \ a \ b \ c \ \text{where}
\]

\[
\text{TransIntrC} :: (\text{Label} \ (c \rightarrow a \rightarrow b)) \rightarrow \text{ArrLangC} \ a \ b \ c
\]
\[
\text{TransArrowC} :: (\text{ArrLangC} \ a \ d \ c) \rightarrow (\text{ArrLangC} \ d \ b \ c) \rightarrow \text{ArrLangC} \ a \ b \ c
\]
\[
\text{TransArgC} :: (\text{Label} \ (c \rightarrow e)) \rightarrow (\text{Label} \ (\text{TransLangC} \ a \ e)) \rightarrow \text{ArrLangC} \ (a \rightarrow b) \ b \ c
\]
\[
\text{TransArgPC} :: (\text{Label} \ (c \rightarrow e)) \rightarrow (\text{Label} \ (\text{TransLangC} \ f \ e)) \rightarrow (\text{Label} \ (f \rightarrow g)) \rightarrow \text{ArrLangC} \ (g \rightarrow b) \ b \ c
\]

\[\text{Codevoorbeeld 5-36}\]

De concrete grammatica uit Codevoorbeeld 5-36 bevat geen argumenten voor extra woorden, leestekens of witruimte. Dit is gedaan omdat de taal erg klein is en geen generische printfunctie voor afgeleid kan worden. Daarom wordt de extra tekst in de printfuncties voor de datatypes toegevoegd en de labels van de gelabelde functies zullen worden geprint. Hiervoor moeten alle gebruikte functies gelabeld worden. De transformatie zelf moet ook gelabeld worden, omdat die namelijk recursief aangeroepen kan worden in de dataconstructoren `TransArgC` en `TransArgPC`.

In Codevoorbeeld 5-37 staat een deel van de transformatiespecificatie van `WhileAST` naar `WhileCST` uitgedrukt in de transformatietaal uit Codevoorbeeld 5-36. Eerst worden de hulppuncties getoond zoals `exprV1_l` en `pOppC_l`. Dit zijn dezelfde functies als uit Codevoorbeeld 5-33 alleen dan gelabeld, daarom hebben de gelabelde functies de suffix `_l` gekregen. De gebruikte dataconstructoren zijn ook van een label voorzien, om zo ook een vriendelijke naam weer te kunnen geven.
pExprV = Label "pExprV" (\x -> case x of
  (ExprV _) -> True
  _         -> False)

exprV1_l = Label "exprv1" (\(ExprV v) -> v)
call1_l  = Label "call1" (\(Call v e) -> v)
call2_l  = Label "call2" (\(Call v e) -> e)

apl_exprV1_l = Label "apl_exprv1" (\(ExprV v) f -> f v)
apl_call1_l = Label "apl_call1" (\(Call v e) f -> f v)
apl_call2_l = Label "apl_call2" (\(Call v e) f -> f e)

oppC_l = Label "OppC" OppC

varTrans_l = Label "varTrans" varTrans
varTrans = TransRuleC (pVar, varTransRule)
           EndRuleC

expr'Trans_l = Label "expr'Trans" expr'Trans
expr'Arrow = TransArgC expr'Arrow exprC_l
expr'Trans = TransRuleC (pExprB, exprBTransRule)
            (TransRuleC (pCall, callTransRule)
            (TransRuleC (pExprV, exprVTransRule)
            (TransRuleC (pExprS, exprSTransRule)
            (TransRuleC (pExprI, exprITransRule)
            (TransRuleC (pBrExpr, exprRTransRule)
            (TransRuleC (pBrExpr, brExprTransRule)
            EndRuleC)))})

exprVTransRule = PutCtorC exprVArrow exprVC_l
exprVArrow = TransArgC exprV1_l varTrans_l
callTransRule = PutCtorC callArrow callC_l
callArrow = TransArrowC (TransIntrC apl_vooruit)
            (TransArrowC (TransIntrC apl_space)
            (TransArrowC (TransArgC call1_l varTrans_l)
            (TransArrowC (TransIntrC apl_openBracket)
            (TransArrowC (TransArgC call2_l exprTrans_l)
            (TransIntrC apl_closeBracket))))

Codevoorbeeld 5-37
printLabel :: (Label a) -> String
printLabel (Label s _) = s

printTransLangC_1 :: (Label (TransLangC a b)) -> String -> String
printTransLangC_1 (Label s x) i = "Transformatie: " ++ s ++ " :=\r\n" ++ printTransLangC x (i++"\t")

printTransLangC :: (TransLangC a b) -> String -> String
printTransLangC (EndRuleC i) = "Einde."
printTransLangC (TransRuleC (x, y) z) i = i ++ "Als invoer voldoet aan " ++ (printLabel x) ++ " dan\r\n" ++ (printTransRuleC y (i++"\t")) ++ "\r\n" ++ (printTransLangC z i)

printTransRuleC :: (TransRuleC a b c) -> String -> String
printTransRuleC (PutCtorC x y) i = i ++ "creeer een " ++ (printLabel y) ++ " op basis van [" ++ (printArrLangC x i) ++ "]"
printTransRuleC (PutCtorEC c) i = i ++ "creeer een " ++ (printLabel c)

printArrLangC :: (ArrLangC a b c) -> String -> String
printArrLangC (TransIntrC x) i = printLabel x
printArrLangC (TransArrowC p q) i = (printArrLangC p i) ++ ", " ++ (printArrLangC q i)
printArrLangC (TransArgC x y) i = "transformeer het resultaat van " ++ (printLabel x) ++ " met " ++ (printLabel y)
printArrLangC (TransArgPC x y z) i = "transformeer het resultaat van " ++ (printLabel x) ++ " met " ++ (printLabel y) ++ " en transformeer die uitvoer met " ++ (printLabel z)

Codevoorbeeld 5-38

printArrLangC :: (ArrLangC a b c) -> String -> String
printArrLangC (TransArrowC p q) i = "plaats: " ++ (printArrLangC p i) ++ " en dan " ++ (printArrLangC q i)
printArrLangC' :: (ArrLangC a b c) -> String -> String
printArrLangC' (TransArrowC p q) i = "pas het resultaat van " ++ (printArrLangC p i) ++ " toe op {" ++ (printArrLangC q i) ++ "}

Codevoorbeeld 5-39
In Codevoorbeeld 5-38 is een printfunctie gegeven voor de transformatietaal. Deze printfunctie voor de transformatietaal is gebaseerd op de pretty-printer uit Sectie 5.1. Daarin is het record met contextinformatie vervangen door een enkele String-instantie. Deze keuze is gemaakt omdat op dit moment alleen gebruik wordt gemaakt van het inspringniveau van de huidige regel. De printer concateneert stukken tekst, labels en recursieve aanroepen aan elkaar.

De uitvoer van de printfunctie is in simpele gestructureerde Nederlandse taal. De lezer moet alleen op de hoogte zijn van de gebruikte functienamen en de notatie voor verschillende stappen van de transformatie. In de taal worden de verschillende stappen geschreven als een soort lijstnotatie, met [ ]-tekens en de stappen worden gescheiden door het woord "en". Het gebruik van deze syntax zorgt ervoor dat de uitvoer beknopt blijft. De volledige uitvoer van de geprinte Expr'-transformatie-instantie, van WhileAST naar WhileCST, is in Codevoorbeeld 5-40 te zien.

```plaintext
Transformatie: expr'Trans :=
    Als invoer voldoet aan pExprB dan
        creer een ExprBC op basis van [apl_exprB1]
    Als invoer voldoet aan pCall dan
        creer een Cal1C op basis van [voer uit, " ", transformeer het resultaat van
            call1 met varTrans, (, transformeer het resultaat van call2 met exprTrans, )]
    Als invoer voldoet aan pExprV dan
        creer een ExprVC op basis van [transformeer het resultaat van exprv1
            met varTrans]
    Als invoer voldoet aan pExprS dan
        creer een ExprSC op basis van [", exprS1, "]
    Als invoer voldoet aan pExprI dan
        creer een ExprIC op basis van [exprv1]
    Als invoer voldoet aan pExprR dan
        creer een ExprRC op basis van [apl_exprR1]
    Als invoer voldoet aan pBrExpr dan
        creer een BrExprC op basis van [(), transformeer het
            resultaat van brExpr1 met exprTrans, ]

Einde.
```

Codevoorbeeld 5-40

De transformatiestappen uit Codevoorbeeld 5-40 zijn technisch gerepresenteerd als een soort van lijstnotatie ([a en b en c]), maar de transformatiestappen kunnen anders gepresenteerd worden. Dit kan door extra woorden te printen in de printArrLangC-functie voor de dataconstructor TransArrowC, zie Codevoorbeeld 5-39. In het laatste voorbeeld zijn extra haakjes gebruikt om de context van de stappen duidelijk te maken. Voor een vergelijking van de uitvoer van de twee printinstanties zie Codevoorbeeld 5-41.

```plaintext
Als invoer voldoet aan pCall dan
    creer een Cal1C op basis van [plaats: voer uit en dan plaats: " ", en dan plaats:
        transformeer het resultaat van call1 met varTrans en dan plaats: ( en dan plaats:
            transformeer het resultaat van call2 met exprTrans en dan ]]

Als invoer voldoet aan pCall dan
    creer een Cal1C op basis van [pas het resultaat van voer uit toe op (pas het resultaat
        van " ", toe op (pas het resultaat van transformeer het resultaat van call1 met varTrans
        toe op (pas het resultaat van ( toe op (pas het resultaat van transformeer het
            resultaat van exprTrans toe op []))))]
```

Codevoorbeeld 5-41

De printfuncties uit Codevoorbeeld 5-38 en Codevoorbeeld 5-39 kunnen naar eigen smaak nog veranderd worden om een andere representatie te geven van de transformatie-instanties. Andere mogelijke aandachtspunten bij de representatie zijn:

- De weergave van functies. Wordt de functienaam geprint zoals die in de programmeertaal geschreven is of wordt er gekozen voor een zinsconstructie van de betekenis van de functie in natuurlijke taal?
- Aansluitend met het vorige aandachtspunt, hoe kunnen gereserveerde woorden en leestekens kenbaar gemaakt worden. Op dit moment wordt de representatie als platte tekst (string-waarde) geprint, maar er is geen verschil tussen woorden die de transformatiestructuur weergeven en woorden die tot de transformatie behoren.
- De volgorde van weergave. Er is gekozen om eerst de constructornaam te printen omdat de constructornaam direct aangeeft wat geconstrueerd gaat worden, terwijl het plaatsen van de dataconstructor in feite de laatste stap van het transformatieproces is.
Hoe is de expliciete volgorde van de te plaatsen argumenten duidelijk kenbaar te maken. Er zijn een drietal mogelijke weergaves gegeven, maar wanneer de drie bovengenoemde aandachtspunten aangepakt worden dan moet in lijn met die oplossing ook de technische functieapplicatie goed weergegeven worden.

In deze sectie is manier getoond om een grammatica met hogere orde argumenten te kunnen representeren als seminauturlijke taal. Hiermee is de transformatie van de transformatiegrammatica naar natuurlijke taal beschreven. Wat overblijft, is de transformatie de andere kant op: van tekst terug naar een instantie van de grammatica. Deze transformatie wordt in de volgende sectie beschreven.

### 5.4.2 Een parser voor de transformatietaal?

Om vanuit de tekstuele representatie naar een instantie van de transformatiegrammatica te komen moet een parser geschreven worden. De parser consumeert tekens en zet die om naar een boomstructuur. Maar alle elementen uit de boomstructuur van de transformatietaal hebben argumenten van hogere orde. Voor de pretty-printer zijn deze functies gelabeld en voor de parser wordt ook uitgegaan van gelabelde functies: de parser moet het label herkennen in de bijbehorende functie opleveren. In deze paragraaf wordt aangetoond waarom het definiëren van een parser voor de transformatietaal weinig zin heeft.

In Sectie 5.4.1 zijn de hogere orde functies voor de elementen uit de transformatietaal van een label voorzien. Dit label kan geparseerd worden en gekoppeld worden met de gelabelde functie met het overeenkomende label. Dit resulteert in een specifieke parser voor elk gelabeld item, zie Codevoorbeeld 5-42. De verschillende instanties kunnen vereenvoudigd worden met een parser voor het type `Label a`. Let hierbij op dat deze parser geschreven is in Haskell, vandaar de iets andere syntaxis ten opzichte van de andere parsers in deze scriptie.

De parsers voor de gelabelde items kunnen gedefinieerd worden met de hulpparser `pLabel`, wanneer deze parsers gedefinieerd zijn kan aan de parsers voor datastructuren van de transformatietaal begonnen worden. Net zoals bij de pretty-printer voor de transformatietaal, is ervoor gekozen om een handgeschreven versie te maken omdat er geen generische functie kan worden afgeleid. Wat geparseerd moet worden wordt bepaald door het op te leveren doeltype. Voor de leesbaarheid zijn extra tekens (woorden) geprint. In de transformatiegrammatica zijn deze extra tekens niet opgenomen als argumenten. Hierdoor moet in de parsers worden gedefinieerd wat geparseerd moet worden.

Voor de concrete grammatica van de transformatietaal kan een set parsers geschreven worden, deze parsers zijn geparameateriseerd om de verschillende gelabelde items uit en grammatica te kunnen parseren. De set parsers bestaat uit een parser voor elk van de dataconstructoren van de transformatietaal, plus een extra parser om een gelabelde transformatie naar een `Label a` te parseren. Voor het gemak is gekozen om een parser te schrijven voor de `1e` variant voor het `TransArrowC`-datatype (zie Codevoorbeeld 5-39). Door eerst een extra woord te parseren wordt linksrecursie vermeden. De uitvoer van de pretty-printer, uit Codevoorbeeld 5-38, voor het `TransArrowC`-datatype is niet te parseren met behulp van parsercombinatoren. Dit komt omdat de parser oneindig doorloopt als gevolg van linksrecursie. De drie hulpparsers voor de grammatica van de transformatietaal zijn weergegeven in Codevoorbeeld 5-43, alle parsers zijn te vinden in Appendix C.
De parsers uit Codevoorbeeld 5-43 parseren precies de uitvoer van de pretty-printer uit Codevoorbeeld 5-38. De parameters van de parsers moeten gevuld worden met de gelabelde functies voor de transformatie van de bronstaal naar de doelstaal of met andere parser-instanties. In Figuur 5-5 zijn de beschrijvingen van de variabelen van de parsers te vinden.

<table>
<thead>
<tr>
<th>Naam argument</th>
<th>Soort invoer</th>
</tr>
</thead>
<tbody>
<tr>
<td>parse_TransLangC</td>
<td>Een parser die een TransLangC-type oplevert.</td>
</tr>
<tr>
<td>f</td>
<td>Prefix / inspringniveau van de regel, een variabele van het type String.</td>
</tr>
<tr>
<td>i</td>
<td>Een gelabelde predicaat.</td>
</tr>
<tr>
<td>parse_TransRuleC</td>
<td>Een parser die een TransRuleC-type oplevert.</td>
</tr>
<tr>
<td>f</td>
<td>Een gelabelde dataconstructor.</td>
</tr>
<tr>
<td>g</td>
<td>Een parser die een TransLangC-type oplevert.</td>
</tr>
<tr>
<td>c</td>
<td>Een parser die een TransLangC-type oplevert.</td>
</tr>
<tr>
<td>i</td>
<td>Prefix / inspringniveau van de regel, een variabele van het type String.</td>
</tr>
<tr>
<td>parse_EndRuleC</td>
<td>Prefix / inspringniveau van de regel.</td>
</tr>
<tr>
<td>parse_PutCtorC</td>
<td>Een gelabelde dataconstructor.</td>
</tr>
<tr>
<td>f</td>
<td>Een parser die een ArrLangC-type oplevert.</td>
</tr>
<tr>
<td>i</td>
<td>Prefix / inspringniveau van de regel, een variabele van het type String.</td>
</tr>
<tr>
<td>parse_PutCtorEC</td>
<td>Een gelabelde dataconstructor.</td>
</tr>
<tr>
<td>f</td>
<td>Een gelabelde functie die op basis van de invoer een nieuw argument introduceert.</td>
</tr>
<tr>
<td>i</td>
<td>Prefix / inspringniveau van de regel, een variabele van het type String.</td>
</tr>
<tr>
<td>parse_TransIntrC</td>
<td>Een parser die een TransArgC-type oplevert.</td>
</tr>
<tr>
<td>f</td>
<td>Een parser die een ArrLangC-type oplevert.</td>
</tr>
<tr>
<td>i</td>
<td>Prefix / inspringniveau van de regel, een variabele van het type String.</td>
</tr>
<tr>
<td>parse_TransArrowC</td>
<td>Een parser die een TransArgC-type oplevert, deze moet matchen met het type van f.</td>
</tr>
<tr>
<td>i</td>
<td>Prefix / inspringniveau van de regel, een variabele van het type String.</td>
</tr>
<tr>
<td>parse_TransArgC</td>
<td>Een gelabelde functie die een argument uit de invoer oplevert.</td>
</tr>
<tr>
<td>i</td>
<td>Prefix / inspringniveau van de regel, een variabele van het type String.</td>
</tr>
<tr>
<td>parse_TransArgPC</td>
<td>Een gelabelde functie die een argument uit de invoer oplevert.</td>
</tr>
<tr>
<td>i</td>
<td>Prefix / inspringniveau van de regel, een variabele van het type String.</td>
</tr>
</tbody>
</table>
De argumenten van de parsers bepalen welke transformatie-instantie wordt geleverd en dus welk bronotype getransformeerd wordt naar het doeltype. Hier begint het nut van de parser wat te vervagen. De parser moet zo gedefinieerd worden dat die de correct getypeerde labels parseert die bij een transformatieregel behoren. Er moet dus aangegeven worden wat de parser mag parseeren: de specifieke labels voor benodigde functies. Zo ontstaat een parser voor elke transformatieregel van de transformatie. Het nut van een parser voor de transformatieset is dus nihil. Het schrijven van een parser is zelfs meer werk dan het schrijven van de transformatiefunctie.

In Codevoorbeeld 5-44 zijn twee parsers te zien voor de transformatiespecificatie van een ExprB naar ExprBC en van Call naar CallC. Deze parsers maken gebruik van de geparameteriseerde parsers uit Codevoorbeeld 5-43. De parsers zijn elk specifiek voor één transformatieregel gedefinieerd. Om een volledige transformatie-instantie te kunnen parseren, bijvoorbeeld voor het datatype Expr' naar ExprC', moet per taallelement een parser beschreven zijn. De parsers voor de taallelementen kunnen samen worden gevoegd tot een parser die het gehele type kan parseren, zie Codevoorbeeld 5-45.

Codevoorbeeld 5-44

De parseExpr'LangInst-parser kan de uitvoer van de pretty-printer uit Codevoorbeeld 5-40 parseren, maar moet de transformatieregels precies in die volgorde tegenkomen waarin de parsers voor de verschillende transformatieregels zijn gedefinieerd. Een nadeel van deze volgorde is dat degene die de transformatie beschrijft op de hoogte moet zijn in welke volgorde de parser de transformatieregels moet tegen komen. Dit betekent dat de parser-instantie bekend moet zijn en de invoer daar precies voor geschreven moet worden. Om de volgorde niet vast te leggen kan met de parsercombinatoren gespeeld worden. Daardoor kan de volgorde van de transformatieregels veranderd worden en kan het aantal transformatieregels variabel gemaakt worden. In Codevoorbeeld 5-46 is een recursieve parser te zien die net zolang verschillende parsers voor de transformatie van Expr' naar ExprC' probeert te parseren totdat een EndRuleC geparseerd wordt. Zo kan bijvoorbeeld de invoer uit Codevoorbeeld 5-47 geparseerd worden door de parser uit Codevoorbeeld 5-46. Maar de werking van de transformatie blijft hetzelfde, door de first-fit-strategie zal altijd de eerst passende regel uitgevoerd worden. Het enige voordeel is dat de volgorde van transformatiedeclaratie nu niet vast ligt door de parser. Een nadeel is echter dat niet volledige parser-instanties ook geparseerd kunnen worden.

Codevoorbeeld 5-46
Transformatie: \(\text{expr'}\text{Trans} ==\)
\begin{equation*}
\text{Als invoer voldoet aan } \text{pBrExpr dan}
\quad \text{creeer een BrExprC op basis van } [\text{plaats: } ( \text{en dan plaats: transformeer het resultaat van } \text{brExpr1 met } \text{exprTrans en dan } )]
\end{equation*}
\begin{equation*}
\text{Als invoer voldoet aan } \text{pBrExpr dan}
\quad \text{creeer een BrExprC op basis van } [\text{plaats: } ( \text{en dan plaats: transformeer het resultaat van } \text{brExpr1 met } \text{exprTrans en dan } )]
\end{equation*}
\begin{equation*}
\text{Als invoer voldoet aan } \text{pExprR dan}
\quad \text{creeer een ExprRC op basis van } [\text{apl_exprR1}]
\end{equation*}
\begin{equation*}
\text{Einde.}
\end{equation*}

**Codevoorbeeld 5-47**

De combinatie van GADTs en gelabelde functies is niet handig voor het schrijven van een parser. De typering van de GADTs en de gelabelde functies moet overeen komen tijdens de creatie van een transformatieregel. Daarnaast is het niet mogelijk om een functie te schrijven die verschillende types op kan leveren op basis van de invoer: het is niet mogelijk om een functie te definiëren van een `String` naar een willekeurig type. Dus een functie die op basis van een naam (de `String`-waarde van het label) de bijbehorende gelabelde functie oplevert is niet mogelijk. Er zijn mogelijkheden om verschillende types op te kunnen leveren, maar deze mogelijkheden vereisen veel handwerk en een vaste typering vooraf. In Codevoorbeeld 5-48 zijn twee mogelijkheden weergegeven waarne verschillende types in één type onder kunnen worden gebracht. Beide mogelijkheden resulteren in een resultaattype waar vooraf alle benodigde types bekend moeten zijn. Deze types kunnen dan in een `MyType` of een boom van `EITHERs` worden verwerkt. De parseerfunctie wordt dan een functie waarin alle mogelijke types van het resultaattype zijn benoemd.

```haskell
:: MyType a b c = C1 a | C2 b | ...
:: EITHER a b = LEFT a | RIGHT b
```

**Codevoorbeeld 5-48**

De parser voor de gelabelde functies moet worden voorzien van een resultaattype waarin alle gelabelde functies zijn gedefinieerd. Alle gelabelde functies moeten dus vooraf gedefinieerd worden en dit resulteert in een typespecifieke parser. Wat betekent dat een specifieke parser geschreven moet worden voor elke transformatie en geen generieke parser\(^7\) ontstaat die hergebruikt kan worden.

Kortom, het is niet gelukt om op een generieke wijze\(^7\) een parseerfunctie te definiëren voor de transformatiegrammatica (GADTs), zonder dat de parseerfunctie met al teveel handwerk gebruikt kan worden voor het parseren van de transformatiedefinities. De parsers die in deze sectie gedefinieerd zijn parseren precies de gedefinieerde transformatiedefinities die in de parser zijn opgenomen en elke transformatiedefinitie moet bekend worden gemaakt binnen de parser. Hierdoor worden de parsers specifieke functies die elk een specifieke transformatieregel kunnen herkennen.

\(^7\) Met een generieke wijze wordt een algemene werkwijze genoemd die gedefinieerd kan worden op grammaticaniveau. De wijze hoeft niet per definitie met generisch programmeren uitgevoerd te worden.
6 Casus: Wetteksten

Om de kracht en de mogelijkheden van de taaltransformator te toetsen is een casus uitgewerkt. De casus is beschikbaar gesteld door de afdeling MDA Services van Capgemini BAS B.V. en gaat over een groot project dat de afdeling voor een overheidsinstantie uitvoert met hun FMDD methode. Als basis van deze casus is de documentatie van het bovengenoemde project gebruikt die uit de Cheetah-tool, de taaltransformator van Capgemini BAS B.V., komt. De domeinabstractie van het wettekstendomein is al door experts gemaakt en is gemaakt vanuit de wetteksten die op dit moment gelden.

Op basis van de documentatie is een abstracte grammatica gedefinieerd met bijbehorende concrete grammatica. Voor de concrete grammatica is een pretty-printer en parser gedefinieerd. Om het geheel executeerbaar te maken is een vertaling naar de While-taal uit Sectie 3 gemaakt. In Figuur 6-1 zijn de verschillende grammatica’s en transformaties schematisch weergegeven.

![Figuur 6-1 - Schematische weergave Casus: Wetteksten](image)

Wanneer men naar Figuur 6-1 kijkt dan valt meteen op dat de helft van het figuur uit de taalhiërarchie van de While-taal bestaat. Voor die hiërarchie zijn de pretty-printer (6) en de parser (7) en de transformaties (8 en 9) al beschreven, namelijk in Sectie 5. De grammatica’s en transformaties voor de While-taal kunnen worden hergebruikt en daarom hoeft alleen aandacht te worden besteed aan de creatie van de abstracte en concrete grammatica voor het Wettekstendomein en de transformaties 1 t/m 5.

6.1 Abstracte grammatica: Wetteksten_{AST}

De abstracte grammatica voor de Wetteksten_{AST} is gebaseerd op veelvoorkomende taalconstructies uit de documentatie. De taal beheerst dus niet de volledige documentatie, maar ontbrekende taalelementen kunnen op dezelfde wijze worden toegevoegd. In Appendix D zijn twee taalinstanties uit de documentatie te zien voor gedefinieerd is.

De Wetteksten_{AST} is een declaratieve taal waarin uitdrukkingen over het Wettekstendomein kunnen worden vastgelegd. Voorbeelden hiervan zijn de declaratie van de partner in het huishouden en het beschrijven van wat moet gebeuren bij het beëindigen van een huishouden. De grammatica kan alleen declaraties beschrijven, maar niet aanroepen. De abstracte grammatica van de taal is te zien in Codevoorbeeld 6-1. In dat codevoorbeeld heeft zowel het datatype als de dataconstructor Declaratie_ een extra _ gekregen, de rede hiervoor is dat het woord declaratie als gereserveerd woord wordt gebruikt in de concrete syntax en daadwoord als type en dataconstructor gedefinieerd wordt.

In de taal zijn declaraties vast te leggen waarin een selectie van een aantal entiteiten te maken is. Op het aantal entiteiten van de selectie wordt een gevalsoverscheiding gemaakt. De gevalsoverscheiding wordt gemaakt op: leeg, één of meerdere elementen. Per gevalsoverscheiding
kunnen toekenning en aanroepen worden gedaan op de entiteiten. De betekenis van de taalelementen is weergegeven in Figuur 6-2.

:: Declaratie_ a b c = Declaratie_(Identifier a) (Selectie a c) (OptioneleToekenning b) (OptioneleToekenning b) (OptioneleToekenning b)
:: Selectie a b = Selectie (Identifier [a]) (BoolExpressie b) (EerstGevonden (Gevallen a))
:: Toekenning a = Uitvoeren (Expressie a) (Toekenning (Identifier a) (Expressie a))
:: Expressie a = Constante String (Verwijzing (Identifier a)) (Toepassen (Identifier a) (Identifier a))
:: BoolExpressie a = Vergelijking (Expressie a) (Expressie a) (EnB (BoolExpressie a) (BoolExpressie a)) (OfB (BoolExpressie a) (BoolExpressie a))
:: AGeval a = AGeval (Identifier [a]) (BoolExpressie a)
:: Identifier a = Identifier String
:: Meerdere a = Enkel a (Meerdere a)
:: Optioneel a = Waarde a (Leeg)
:: OptioneleToekenning a = Optioneel (Toekenningen a)
:: Toekenningen a = Meerdere a
:: Gevallen a = Meerdere (AGeval a)


<table>
<thead>
<tr>
<th>Naam type</th>
<th>Betekenis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaratie_ a b c</td>
<td>Het Declaratie_-element is het hoogste element uit de taal en heeft ruimte voor zijn naam en type, de selectie van entiteiten en voor de drie gevallensorderscheidingen: leeg, één of meerdere elementen. De naam heeft een _-teken erbij gekregen omdat het type en de dataconstructor Declaratie beide voorkomen als non-terminal en terminal in de concrete syntaxis.</td>
</tr>
<tr>
<td>Selectie a b</td>
<td>Het Selectie-element heeft de keuze voor een selectie binnen een lijst of een selectie uit meerdere gevallen.</td>
</tr>
<tr>
<td>Toekenning a</td>
<td>Het Toekenning-element voert een expressie uit of het kent een waarde toe aan een Identifier.</td>
</tr>
<tr>
<td>Expressie a</td>
<td>Het Expressie-element is een &quot;programmeerelement&quot;, het is ofwel een constante, een verwijzing naar een Identifier of toepassing van twee Identifiers op elkaar.</td>
</tr>
<tr>
<td>BoolExpressie a</td>
<td>Het BoolExpressie-element is een wiskundig element wat een vergelijking tussen twee Expressie-a-elementen is of de logische en of of van twee BoolExpressions op.</td>
</tr>
<tr>
<td>AGeval a</td>
<td>Het AGeval-element heeft de ruimte voor een Identifier en een BoolExpression over die Identifier. Als het resultaat van de BoolExpression een waar-waarde oplevert zal de Identifier de eerstgevonden waarde worden van de selectie. De naam heeft een extra _ gekregen omdat het type en de dataconstructor AGeval beide voorkomen als non-terminal en terminal in de concrete syntaxis.</td>
</tr>
<tr>
<td>Identifier a</td>
<td>Het Identifier-element geeft een naam aan een getypeerde expressie of instantie.</td>
</tr>
<tr>
<td>Meerdere a</td>
<td>Het Meerdere-element heeft de keuze voor één argument of voor een argument en een verwijzing naar een element van zijn eigen type.</td>
</tr>
<tr>
<td>Optioneel a</td>
<td>Het Optioneel-element heeft de keuze tussen één argument of geen argument.</td>
</tr>
</tbody>
</table>

Figuur 6-2 - Betekenis van de taalelementen WettekstenAST
In Codevoorbeeld 6-2 is het partner in huishouden voorbeeld en in Codevoorbeeld 6-3 is het beëindigen huishouden voorbeeld te zien, beide zijn gemodelleerd in WettekstenAST. De oorspronkelijke specificaties staan in Appendix D. De abstracte syntax is moeilijk te lezen, maar in de volgende sectie wordt de concrete syntaxis beschreven die deze syntaxis decoreert met het doel dat de uitvoer leesbaar wordt.

Statement (Identifier "partnerrol") (Selectie (Identifier "huishouden") (Vergelijking (Verwijzing (Identifier "huishoudenDeelnameRol")) (Constante "Partner"))) Waarde (Enkel (Toekenning (Identifier "partnerInHuishouden") (Verwijzing (Identifier "partnerrolBurger")))))

Codevoorbeeld 6-2 - Het partner in huishouden voorbeeld uit Appendix D in WettekstenAST

Statement (Identifier "teBeeindigenHuishouden") (Selectie (Identifier "huishouden") (Vergelijking (Verwijzing (Identifier "huishoudenAanvrager")) (Verwijzing (Identifier "burger"))) (Waarde (Meer (Uitvoeren (Toepassen (Identifier "attenderen achterblijvende partner") (Identifier "teBeeindigenHuishouden"))) (Enkel (Uitvoeren (Toepassen (Identifier "beëindigGeldigheidHuishouden") (Identifier "teBeeindigenHuishouden"))))) Leeg Leeg

Codevoorbeeld 6-3 - Het beëindigen huishouden voorbeeld uit Appendix D in WettekstenAST

6.2 Concrete grammatica: WettekstenCST

Om de abstracte grammatica uit Sectie 6.1 leesbaar te maken is de grammatica gedecoreerd met extra woorden, lenteekens en witruimte. Het decoreren is op dezelfde wijze gedaan als voor de While-taal. Voor elk element uit WettekstenAST is een gedecoreerd element in WettekstenCST gemaakt. De extra woorden die grammatica-instanties leesbaar maken zijn toegevoegd aan de grammatica. De WettekstenCST-grammatica is hieronder in Codevoorbeeld 6-4 te zien. In die grammatica is gebruik gemaakt van een overkoepelende set leestekenelementen, die elementen zijn te zien in Codevoorbeeld 6-5.
:: Blok = Blok NewLine a NewLine
:: Lidwoord = De | Het | Een
:: Is = Is
:: Met = Met
:: Indien = Indien
:: Uniek = Uniek
:: Niet = Niet
:: Gevonden = Gevonden
:: Einde = Einde
:: Declaratie = Declaratie
:: Colon = Colon // :
:: Apostrophe = Apostrophe // '
:: OpenBracket = OpenBracket // (
:: CloseBracket = CloseBracket // )
:: OpenSquareBracket = OpenSquareBracket // [
:: CloseSquareBracket = CloseSquareBracket // ]
:: Dash = Dash // -
:: Dot = Dot // .
:: Comma = Comma // ,
:: Semicolon = Semicolon // ;
:: Quote = Quote // "
:: PlusSign = PlusSign // +
:: MinusSign = MinusSign // -
:: MultiSign = MultiSign // *
:: DivSign = DivSign // /
In Codevoorbeeld 6-6 is het partner in huishouden voorbeeld en in Codevoorbeeld 6-7 het beëindigen huishouden voorbeeld in Wetteksten\textsubscript{CST} te zien. De oorspronkelijke specificaties staan in Appendix D. Deze representaties zijn niet met de hand geschreven maar getransformeerd vanuit de abstracte syntax uit Codevoorbeeld 6-2 en Codevoorbeeld 6-3 met de transformatie die beschreven wordt in Sectie 6.5. De onderstaande declaraties zijn evenmin te begrijpen. Maar in de volgende sectie wordt de pretty-printer voor Wetteksten\textsubscript{CST} beschreven die de twee declaraties naar tekst transformeert en leesbaar maakt.

Declaratie\textsubscript{n} De Space (Identifier\textsubscript{n} "partnerrol") Space (Selectie\textsubscript{n} Is Space De Space (Identifier\textsubscript{n} "huishouden") Space Met Space (Vergelijking\textsubscript{n} (Verwijzing\textsubscript{n} (Identifier\textsubscript{n} "huishoudenDeelnameRol")) Space Is Space (Constante\textsubscript{n} "Partner"))) NewLine Indien Space Gevonden (Blok NewLine (Waarde\textsubscript{n} (Enkel\textsubscript{n} (Toekenning\textsubscript{n} De Space (Identifier\textsubscript{n} "partnerinHuishouden") Space Is Space (Verwijzing\textsubscript{n} (Identifier\textsubscript{n} "partnerrolBurger")))))

NewLine) Indien Space Niet Space Gevonden (Blok NewLine Leeg\textsubscript{n} NewLine) Indien Space Niet Space Uniek (Blok NewLine Leeg\textsubscript{n} NewLine) Einde Space Declaratie

Codevoorbeeld 6-6 - Het partner in huishouden voorbeeld uit Appendix D in Wetteksten\textsubscript{CST}

Declaratie\textsubscript{n} De Space (Identifier\textsubscript{n} "teBeeindigenHuishouden") Space (Selectie\textsubscript{n} Is Space De Space (Identifier\textsubscript{n} "huishouden") Space Met Space (Vergelijking\textsubscript{n} (Verwijzing\textsubscript{n} (Identifier\textsubscript{n} "HuishoudenAanvrager")) Space Is Space (Verwijzing\textsubscript{n} (Identifier\textsubscript{n} "burger"))))) NewLine Indien Space Gevonden (Blok NewLine (Waarde\textsubscript{n} (Meer\textsubscript{n} (Uitvoeren\textsubscript{n} Voer Space Uit Space (Toepassen\textsubscript{n} Pas Space Toe Space (Identifier\textsubscript{n} "attenderenAchterblijvende partner") Space Bij Space (Identifier\textsubscript{n} "teBeeindigenHuishouden")) NewLine (Enkel\textsubscript{n} (Uitvoeren\textsubscript{n} Voer Space Uit Space (Toepassen\textsubscript{n} Pas Space Toe Space (Identifier\textsubscript{n} "beeiindigGeldigheidHuishouden") Space Bij Space (Identifier\textsubscript{n} "teBeeindigenHuishouden")))))) NewLine) Indien Space Niet Space Gevonden (Blok NewLine Leeg\textsubscript{n} NewLine) Indien Space Niet Space Uniek (Blok NewLine Leeg\textsubscript{n} NewLine) Einde Space Declaratie

Codevoorbeeld 6-7 - Het beëindigen huishouden voorbeeld uit Appendix D in Wetteksten\textsubscript{CST}

\textbf{6.3 Een pretty-printer voor Wetteksten\textsubscript{CST}}

In vorige sectie zijn twee instanties van Wetteksten\textsubscript{CST} gegeven. De twee instanties zijn moeilijk leesbaar en alleen door goed te kijken kunnen herkenningspunten worden gezien. Om instanties van Wetteksten\textsubscript{CST} naar tekst te transformeren worden instanties van de pretty-printer beschreven en afgeleid. De pretty-printer is transformatie #1 uit Figuur 6-1. De volledige pretty-printer voor Wetteksten\textsubscript{CST} in te zien in Codevoorbeeld 6-8.

derive gPrettyPrint Identifier\textsubscript{n}, Declaratie\textsubscript{n}, Selectie\textsubscript{n}, Toekenning\textsubscript{n}, Expressie\textsubscript{n}, BoolExpressie\textsubscript{n}, AGeval\textsubscript{n}, Meerdere\textsubscript{n}, Optioneel\textsubscript{n}
derive gPrettyPrint Lidwoord, Is, Met, Indien, Uniek, Niet, Gevonden, Einde, Declaratie, Eerst, Van, Pas, Toe, Bij, En, Of, Geval, Voer, Uit
gPrettyPrint (\{Blok\}) f ctx (Blok a b c) = newline +
\{f (ctx & cindent = ctx.cindent +++ "\t") b +++ newline\}

Codevoorbeeld 6-8 – Pretty-print-instantie voor Wetteksten\textsubscript{CST}

De pretty-printer voor Wetteksten\textsubscript{CST}, uit Codevoorbeeld 6-8, heeft maar één met de hand gedefinieerde functie-instantie. In die instantie wordt het inspringniveau van de regels vergroot binnen een Blok, dit ten behoeve van de opmaak. Alle andere functie-instanties kunnen worden afgeleid. Om het juiste inspringniveau te printen moeten de dataconstructoren die moeten inspringen nog kenbaar worden gemaakt aan de pretty-print-instantie voor het type CONS. In dit geval zijn dat de dataconstructoren Toekenning\textsubscript{n} en Uitvoeren\textsubscript{n} van het type Toekenning\textsubscript{n} a.

In Codevoorbeeld 6-9 en Codevoorbeeld 6-10 zijn de tekstrepresentaties te zien na het pretty-printen van de declaraties uit Codevoorbeeld 6-6 en Codevoorbeeld 6-7. De twee onderstaande declaraties
zijn voor eenieder leesbaar die de Nederlandse taal beheert. Het enige waar onduidelijkheid over kan ontstaan zijn de gebruikte namen voor de identifiers. Deze namen zijn in camelCase geschreven.

De partnerrol Is De huishouden Met huishoudenDeelnameRol Is Partner
Indien Gevonden
    De partnerInHuishouden Is partnerrolBurger
Indien Niet Gevonden

Indien Niet Uniek
Einde Declaratie
Codevoorbeeld 6-9 - Het partner in huishouden voorbeeld uit Appendix D na pretty-print van WettekstenCST

De teBeeindigenHuishouden Is De huishouden Met HuishoudenAanvrager Is burger
Indien Gevonden
    Voer Uit Pas Toe attenderenAchterblijvendePartner Bij teBeeindigenHuishouden
    Voer Uit Pas Toe beeindigGeldigheidHuishouden Bij teBeeindigenHuishouden
Indien Niet Gevonden

Indien Niet Uniek
Einde Declaratie
Codevoorbeeld 6-10 - Het beëindigen huishouden voorbeeld uit Appendix D na pretty-print van WettekstenCST

6.4 Een parser voor WettekstenCST

De inverse van de pretty-printer is de parser en voor de parser moet dezelfde handeling worden gedaan als voor de pretty-printer. Dit betreft de handmatige functie-instantie voor het datatype Blok a. Daarnaast moeten voor de types met dataconstructoren die terminalwaardes representeren parser-instanties worden geschreven. Deze parsers kunnen worden gedefinieerd met behulp van de hulpparser pToken. Het type Lidwoord is daar nog een kleine uitzondering op, dat type kan namelijk drie terminalwaardes representeren. Voor alle andere datatypes kan de gParse functie worden afgeleid. De parser is transformatie #2 uit Figuur 6-1. In Codevoorbeeld 6-11 zijn de belangrijkste declaraties van de gParse functie voor WettekstenAST te zien.

derive gParse Identifier_n, Declaratie_n, Selectie_n, Toekenning_n, Expressie_n, IntExpressie_n, BoolExpressie_n, AGeval_n, Meerdere_n, Optioneel_n
gParse{|Blok|} f ctx  = (pToken (fromString newline) NewLine) /
    (f {ctx & cindent = ctx.cindent +++ \"\t\"}) /
    (pToken (fromString newline) NewLine) /
    yield (Blok a b c)
gParse{|Lidwoord|} _   = (pToken "De" De)
    \!/ (pToken "Het" Het)
    \!/ (pToken "Een" Een)
gParse{|Is|} _   = pToken "Is" Is
Codevoorbeeld 6-11

6.5 Transformatie tussen WettekstenAST en WettekstenCST

De declaraties uit Codevoorbeeld 6-7 en Codevoorbeeld 6-8 zijn natuurlijk niet met de hand gedefinieerd, maar getransformeerd vanuit de abstracte syntax (uit Codevoorbeeld 6-6 en Codevoorbeeld 6-7). Deze transformatie is gedefinieerd in de transformatietaal uit Sectie 5.4 en is een decorerende transformatie.

In Codevoorbeeld 6-12 is een deel van de transformatie-instantie van WettekstenAST naar WettekstenCST te zien, dat is transformatie #4 uit Figuur 6-1. Het betreft de decoratie van Toekenningen en


8 In camelCase worden samengestelde woorden of zinnen aan elkaar geschreven, waar alle woorden behalve het eerste woord met een hoofdletter beginnen.
Identifiers. De volledige transformatiespecificatie heeft een transformatieregel voor elke dataconstructor en een Arrow voor elke argument van de dataconstructoren.

toekenning2Toekenning_n = TransRule (pToekenning, toekenning2Toekenning_nRule) (TransRule (pUitvoeren, uitvoeren2Uitvoeren_nRule) EndRule)
toekenning2Toekenning_nRule = PutCtor toekenning2Toekenning_nArrow Toekenning_n
toekenning2Toekenning_nArrow = TransArrow (TransIntr (\_ \rightarrow \{f \rightarrow f \) De})
(TransArrow (TransIntr (\_ \rightarrow \{f \rightarrow f \) Space}))
(TransArrow (TransArg (\(Toekenning\ a\ b) \rightarrow a\) identifier2Identifier_n)
(TransArrow (TransArg (\(Toekenning\ a\ b) \rightarrow a\) Space))
(TransArrow (TransArg (\(Toekenning\ a\ b) \rightarrow a\) Is))
(TransArrow (TransArg (\(Toekenning\ a\ b) \rightarrow a\) Space)))
(TransArrow (TransIntr (\_ \rightarrow \{f \rightarrow f \) Space))
(TransArrow (TransArg (\(Toekenning\ a\ b) \rightarrow a\) Space))
(TransArrow (TransArg (\(Toekenning\ a\ b) \rightarrow a\) Space))))
uitvoeren2Uitvoeren_nRule = PutCtor uitvoeren2Uitvoeren_nArrow Uitvoeren_n
uitvoeren2Uitvoeren_nArrow = TransArrow (TransIntr (\_ \rightarrow \{f \rightarrow f \) Voer})
(TransArrow (TransIntr (\_ \rightarrow \{f \rightarrow f \) Space))
(TransArrow (TransArg (\(Uitvoeren\ a) \rightarrow a\) Space)))
(TransArrow (TransArg (\(Uitvoeren\ a) \rightarrow a\) Space))
(TransArrow (TransArg (\(Uitvoeren\ a) \rightarrow a\) Space))
identifier2Identifier_n = TransRule (noCheck, identifier2Identifier_nRule) EndRule
identifier2Identifier_nRule = PutCtor identifier2Identifier_nArrow Identifier_n
identifier2Identifier_nArrow = TransIntr (\x \rightarrow \{f \rightarrow f\ (\(Identifier\ a) \rightarrow a\) x))

Codevoorbeeld 6-12

De transformatie terug, van WettekstenCST naar WettekstenAST, kan op dezelfde wijze gedefinieerd worden. De de-decoratie van Toekenningen en Identifiers is weergegeven in Codevoorbeeld 6-13. Deze transformatie is #3 uit Figuur 6-1.

toekenning_n2Toekenning = TransRule (pToekenning', toekenning_n2ToekenningRule)
toekenning_n2ToekenningRule = PutCtor toekenning_n2ToekenningArrow Toekenning_n
toekenning_n2ToekenningArrow = TransArrow (TransArg (\(Toekenning\ n\ _\ _\ _\ _\ _\ _\ _\ e) \rightarrow i)
(identifier_n2Identifier)
(TransArg (\(Toekenning\ n\ _\ _\ _\ _\ _\ _\ _\ e) \rightarrow e)
expressie_n2Expressie)
uitvoeren_n2UitvoerenRule = PutCtor uitvoeren_n2UitvoerenArrow Uitvoeren_n
uitvoeren_n2UitvoerenArrow = TransArg (\(Uitvoeren\ n\ _\ _\ _\ _\ _\ _\ _\ e) \rightarrow e)
expressie_n2Expressie
identifier_n2Identifier = TransRule (noCheck, identifier_n2IdentifierRule) EndRule
identifier_n2IdentifierRule = PutCtor identifier_n2IdentifierArrow Identifier_n
identifier_n2IdentifierArrow = TransIntr (\x \rightarrow \{f \rightarrow f\ (\(Identifier\ n\ a) \rightarrow a\) x))

Codevoorbeeld 6-13

6.6 Transformatie van Wetteksten\textit{AST} naar While\textit{CST}

De vorige transformaties zijn allemaal binnen hetzelfde domein gebleven namelijk het Wettekstendomein. In Figuur 6-1 transformeert transformatie #5 de kennis uit het Wettekstendomein naar het technische domein van de softwareontwikkelaars. Deze transformatie is lastiger dan de transformatie tussen Wetteksten\textit{AST} en Wetteksten\textit{CST}. Er is namelijk geen één op één relatie tussen de Wetteksten\textit{AST} en While\textit{CST}.

De transformatie van Wetteksten\textit{AST} naar While\textit{AST} is gedefinieerd in de transformatietaal. De transformatieregels voor dezelfde types uit Codevoorbeeld 6-12 zijn weergegeven in Codevoorbeeld 6-14. Voor het doelplatform wordt ervan uitgegaan dat er een soort van query-tool/object is wat entiteiten retourneert op basis van het meegegeven type. Het object kan bij het uitvoeren een restrictieparameter hebben die de selectie van de entiteiten bepaald.
Het resultaat na de transformatie #8 en #6 uit Figuur 6-1 is weergegeven in Codevoorbeeld 6-15 en Codevoorbeeld 6-16. In beide voorbeelden is te zien dat in het `IfC`-statement een leeg statement is geprint. Het lege statement wordt door de transformatie toegevoegd. In de transformatie is het vanwege de verwijderde linksrecursie makkelijker om eerst een leeg statement te plaatsen waarna altijd een instantie van het type `Stmt` kan worden geplaatst.

In Codevoorbeeld 6-16 is twee keer een toekenning gedaan aan `VOID`. De rede hiervoor is dat in de `While`-taal geen ruimte is voor een procedureaanroep zonder het resultaat op te vangen. De toekenning aan `VOID` kan worden gezien als het niet hebben of het negeren van het resultaat.

Proc partnerInHuishouden (huishouden)
Begin
  lQuery := Voer uit QueryCreate(huishouden).
  lResult := Voer uit lQuery(huishoudenDeelnameRol=="Partner").
  Als lResult.Size()==1 Dan
    .
    partnerInHuishouden := lResult
  Anders
    Als lResult.Size()==0 Dan
    .
    End Als
  End Als
End

Codevoorbeeld 6-15 - Het partner in huishouden voorbeeld uit Appendix D na pretty-print van WhileCST

Proc teBeeindigenHuishouden (burger)
Begin
  lQuery := Voer uit QueryCreate(huishouden).
  lResult := Voer uit lQuery(huishoudenAanvraag==burger).
  Als lResult.Size()==1 Dan
    .
    VOID := Voer uit attenderenAchterblijvendePartner(lResult).
    VOID := Voer uit beëindigGeldigheidHuishouden(lResult)
  Anders
    Als lResult.Size()==0 Dan
    .
    End Als
  End Als
End

Codevoorbeeld 6-16 - Het beëindigen huishouden voorbeeld uit Appendix D na pretty-print van WhileCST
6.7 Voorbeeldtransformatie van de Wettekstentaal naar de While-taal

De twee voorbeelden uit Appendix D die in dit hoofdstuk zijn gebruikt, zijn gedefinieerd in WettekstenAST waarna de verschillende transformaties erop zijn uitgevoerd. In deze laatste sectie wordt de taaltransformator gebruikt vanuit de gebruikersrol, in dit geval de belastingsexpert. Als voorbeeld wordt Codevoorbeeld 6-17 gebruikt.

De relevante besturing is de Besturing betaaladviezen met
Besturing betaaladviezen.toeslagregeling gelijk aan regeling
Indien gevonden
Het laatste vrijgavejaar is relevante besturing.vrijgavejaar
De laatste vrijgavemaand is relevante besturing.vrijgavemaand
De datum afgifte vrijgave is relevante besturing.datum afgifte
Indien niet gevonden
Het laatste vrijgavejaar is 2005
De laatste vrijgavemaand is 1
De datum afgifte vrijgave is de eerste dag van het jaar van laatste vrijgavejaar
Indien geval niet uniek
Einde declaratie

Codevoorbeeld 6-17 - 2.4 laatst vrijgegeven betaalmaand uit Beschrijving van businessconcept Bepalen Betaaladvies van 2009-01

Wanneer de code uit Codevoorbeeld 6-17 herschreven wordt naar de syntaxis van WettekstenCST ontstaat de code die weergegeven is in Codevoorbeeld 6-18. Die code kan geparseerd worden en zo ontstaat een WettekstenCST-syntaxisboom die weergegeven is in Appendix E in Codevoorbeeld 6-1. In dat appendix zijn ook alle tussenresultaten van de verschillende transformaties te zien beginnend bij de definitie uit Codevoorbeeld 6-17. In Codevoorbeeld 6-19 is de uiteindelijke representatie in tekstuele representatie in WettekstenCST tot een tekstuele representatie in WhileCST te komen.

De relevante Besturing Is De besturingBetaalAdviezen Met BesturingBetaalAdviezenToeslagregeling Is Regeling
Indien gevonden
De laatste Vrijgavejaar Is relevanteBesturingVrijgavejaar.
De laatste Vrijgavemaand Is relevanteBesturingVrijgavemaand.
De datum Afgifte Vrijgave Is relevanteBesturingDatumAfgifte
Indien niet gevonden
De laatste Vrijgavejaar Is tweeDuizendVijf.
De laatste Vrijgavemaand Is een.
De datum Afgifte Vrijgave Is eersteDagVanHetJaarVanHetLaatsteVrijgavejaar
Indien niet uniek
Einde Declaratie

Codevoorbeeld 6-18 – Definitie uit Codevoorbeeld 6-17 in tekstuele representatie WettekstenCST

Proc relevanteBesturing (besturingBetaalAdviezen)
Begin
lQuery := Voer uit Query.Create(besturingBetaalAdviezen).
lResult := Voer uit lQuery(BesturingBetaalAdviezenToeslagregeling==Regeling).
Als lResult.Size()==1 Dan
laatsteVrijgavejaar := relevanteBesturingVrijgavejaar.
laatsteVrijgavemaand := relevanteBesturingVrijgavemaand.
dataumAfgifteVrijgave := relevanteBesturingDatumAfgifte
Anders
Als lResult.Size()==0 Dan
laatsteVrijgavejaar := tweeDuizendVijf.
laatsteVrijgavemaand := een.
dataumAfgifteVrijgave := eersteDagVanHetJaarVanHetLaatsteVrijgavejaar
Anders
Einde Als
End Als

End Codevoorbeeld 6-19 – Definitie uit Codevoorbeeld 6-17 in tekstuele representatie WhileCST
In Codevoorbeeld 6-19 is de uiteindelijke technische code te zien als geprinte WhileCST-taal. De volledige transformatie van de tekstuele representatie in WettekstenCST naar de tekstuele representatie van WhileCST gebruikt vijf stappen (de nummers tussen haakjes zijn de transformatienummers uit Figuur 6-1):

1. Parseren, van tekst naar WettekstenCST (#2).
2. De-decoratie, van WettekstenCST naar WettekstenAST (#4)
3. Syntaxistransformatie, van WettekstenAST naar WhileAST (#5)
4. Decoratie, van WhileAST naar WhileCST (#8)
5. Pretty-printen, van WhileCST naar tekst (#6)

De volledige transformatie is weergegeven Codevoorbeeld 6-20 in pseudo functionele code. Hierbij moet worden gezegd dat de parser niet direct een grammatica-instantie oplevert, maar een lijst met mogelijke parseerresultaten met de restinvoer. De gebruikte gParse-functie is deterministisch, dus de lijst is leeg of bevat één element.

```
WettekstenTekst2WhileTekst invoer = (gPrettyPrint' o (transEval decl2DeclC) o (transEval statement2Decl) o (transEval declaratie_n2Statement) o gParse') invoer
```

Codevoorbeeld 6-20

De functie WettekstenTekst2WhileTekst transformeert invoer die voldoet aan WettekstenCST naar tekst die voldoet aan WhileCST. Deze functie is in het prototype niet mogelijk door de scheiding van de transformatietaal in Haskell en grammaticadefinitie in Clean.
7 Gerelateerd werk

In dit hoofdstuk worden een aantal onderzoeksprojecten behandeld die ingezet kunnen worden als oplossing of deeloplossing voor het communicatieprobleem. De taaltransformator uit dit onderzoek bestrijkt twee onderzoeksgebieden namelijk de presentatie aan de gebruiker en de vertaling naar een andere representatie. In de literatuur is een duidelijke scheiding aanwezig tussen die twee onderzoeksgebieden. Daarom worden de twee gebieden apart behandeld. Dit hoofdstuk eindigt met twee tools die zowel grammaticapresentatie en grammaticatransformatie in zich hebben.

7.1 Taaltransformatie en compilatie

In de introductie is gezegd dat het onderzoek dicht bij de compilerwereld ligt, maar niet voldoende aansloot bij het probleem van dit onderzoek. In het onderzoek zijn wel methodes gebruikt die afstammen uit de compilerwereld.

Een belangrijk deel van een compiler is de parser die zinsstructuren herkent en omzet naar een boomstructuur. Twee tools die veel worden gebruikt in de compilerwereld om parsers te genereren zijn Yacc (Yet Another Compiler Compiler) [11] en Bison [8]. In deze sectie wordt niet ingegaan op de verschillen tussen de tools, maar wordt op de achterliggende gedachte en technieken ingegaan. Beide tools gebruiken een BNF-stijl grammatica als invoer en creëren een programma in C of C++ code dat tekst kan herkennen en een boomstructuur oplevert. Op de boomstructuur kunnen semantische acties worden gedefinieerd die worden uitgevoerd zodra een passende nodestructuur ontstaat. Het gegenereerde programma is meer dan een parser alleen, het compileert en herschrijft ook de uitvoer.

Het verschil tussen onze transformator en de Yacc/Bison compilers is dat in de taaltransformator uit deze scriptie het parseren en herschrijven gescheiden is. Het herschrijven is in een aparte transformatie beschreven. Het doel en idee van de parsers is wel hetzelfde: beide methodes genereren op basis van een grammatica de parser. De strategie achter de parser is echter wel verschillend. Een Yacc of Bison parser gebruikt een parsetabel waarin wordt aangegeven welke grammaticaregel moet worden uitgevoerd in een bepaalde status bij een bepaald karakter. Yacc en Bison gebruiken een LR strategie (rechts afleidende). De generische parser is een recursieve top-down parser (Recursive Decent Parser) die recursief functie-instanties aantrekt en zo de structuur van de grammatica volgt. Een recursieve top-down parser gebruikt een LL strategie (links afleidende). De verschillende strategieën resulteren in andere klassen van talen die geparseerd kunnen worden. Yacc en Bison hebben met hun LR strategie geen last van linksrecursie. De taaltransformator staat ongeveer gelijk aan Yacc of Bison, de tool neemt een grammatica als invoer en levert een functie of programma op die tekst vertaalt naar een boomstructuur. Yacc en Bison zijn alleen bedoeld voor het herschrijven van de invoer naar uitvoerbare code, terwijl de taaltransformator de zinstructuur intact houdt en transformaties in een later stadium toelaat. Yacc en Bison parsers/compilers zijn meer geschikt voor het herschrijven van statische talen. In dit onderzoek zou een Yacc of Bison compiler geschreven kunnen worden voor de transformatie van de ASTs of voor een specifieke CST naar AST. De compositie van transformaties uit dit onderzoek is ook mogelijk met Yacc en Bison, alleen moeten de interne syntaxisbomen steeds geprint en geparseerd worden.

In het onderzoek is voor een modelleeraanpak gekozen. In de modelleerwereld zijn verschillende tools die modelltransformaties mogelijk maken. In de inleiding is te zien dat de terminologie uit modelleerwereld te transformeren is naar de programmeerwereld en daarom kunnen de technieken uit de modelleerwereld gebruikt worden voor dit onderzoek. Een modelltransformatie kan worden gezien als een grammaticatransformatie. In deze paragraaf wordt de terminologie uit de modelleerwereld gebruikt. De Object Management Group (OMG) heeft in 2003 een concept voor Query/View/Transformation (QVT) opgeleverd [23]. Voorafgaand aan die aanbeveling zijn aantal andere varianten met een zelfde
soort methodiek gemaakt [12, 17, 23]. De QVT-achtige modeltransformatoren gebruiken drie elementen: queries, views en transformaties. Een query levert een aantal objecten uit het model op, een view is een representatie binnen het model en een transformatie koppelt twee views elkaar en zet waarden over van de bron naar het doel.

Transformaties zijn beschreven in een eigen taal die geïntegreerd is een hosttaal, bij Tefkat is dat bijvoorbeeld Java. Hierdoor kan in de transformatieregels gebruik gemaakt worden van de hosttaal. QVT-achtige modellen zijn gebaseerd op een metamodel-architectuur, veelal de Meta Object Facility (MOF). De modeltransformatoren gebruiken XMI (XML Metadata Interchange) als invoer. XMI is het metamodel voor modeldefinities. Als een QVT-achtige modeltransformatie gebruikt was voor dit onderzoek, dan hadden de grammatica’s en ook de grammaticadefinitie in XMI gedefinieerd moeten worden.

Als laatste zijn er nog verschillende taaltransformatietools die een zelfde splitsing hebben als de taaltransformateur uit dit onderzoek namelijk een scheiding tussen grammaticadefinitie en grammaticatransformatie/termherschrijving. Twee voorbeelden van dergelijke tools zijn ASF+SDF [5] en Stratego/XT [28].

ASF+SDF gebruikt het Syntax Definition Formalism (SDF) waarin de grammatica gedefinieerd kan worden. Op een soortgelijke manier als Yacc wordt een parser gegenereerd die alle mogelijke boomstructuren oplevert, de parser is dus niet deterministisch. Een filter kan alle irrelevante bomen uit het resultaat filteren. De boomstructuur bestaat uit ATerms, wat geannoteerde datastructuren zijn. Om de syntaxisboom te transformeren wordt het Algebraic Specification Formalism (ASF) gebruikt. ASF is een notatie om herschrijfregels in te definiëren.

Stratego/XT maakt net als ASF+SDF gebruik van SDF voor de grammaticadefinitie. Voor het herschrijven van de syntaxisboom wordt de taal Stratego gebruikt en een set transformatietools die XT wordt genoemd. Beide tools maken gebruik van dezelfde grammaticadefinitietool, maar gebruiken een ander herschrijfsysteem. In beide tools kan de gecreëerde syntaxisboom worden geprint met een zogenaamde unparser (ofwel pretty-printer). Deze twee tools komen het beste overeen met de gecreëerde taaltransformator.

7.2 Informatiepresentatie

Om informatie of kennis aan gebruiker te presenteren zijn natuurlijke talen gebruikt. De taaltransformatoor en tools zoals ASF+SDF en Stratego/XT gebruiken een pretty-printer om de informatie te presenteren. In alle drie de tools zijn zowel de invoer als de uitvoer in ASCII tekst. De gebruiker kan deze lezen of bewerken in een willekeurige tekstverwerker.

In de introductie is de afweging tussen grafische en tekstuele modellen behandeld en de keuze is gevallen op tekstuele modellen. Maar in sommige gevallen is het handig om meer presentatiemogelijkheden te hebben dan tekst alleen.


De tekstuele invoer en uitvoer van de taaltransformatoor kan door Proxima worden voorzien van extra opmaak. Zo kunnen koppen worden voorzien van een groter of vetter font, kunnen er lijnen getekend worden die scheidingen aangeven of kunnen wiskundige formules opgemaakt worden in wiskundige representatie. Deze toevoeging zal de uitvoer voor de gebruiker duidelijker maken. Door een tool als Proxima kan de presentatiegrammatica nog beter aansluiten bij het gebruikte jargon.

7.3 Taalcreatie en taaltransformatie

De tool Cheetah is een taaltransformator waarin een grammatica gemodelleerd kan worden. De grammatica kan in verschillende niveaus worden gedefinieerd, waardoor de verschillende niveaus her te gebruiken zijn. Op basis van de grammatica wordt door de tool een editor getoond waarmee de gebruiker op een gestructureerde manier de grammatica kan invullen. Naast de editor kan de tool door middel van transformatoren de gemaakte zinnen/definities transformeren naar het gewenste doelplatform. De uitvoer van de transformator is tekst. Het is niet mogelijk om tussen twee grammatica’s te transformeren. Er zijn wel verschillende representaties mogelijk als uitvoer na een transformatie zoals executeerbare code of documentatie.

8 Conclusie

In deze scriptie is aangetoond dat verschillende transformaties, zoals tussen syntaxisbomen en tekst en tussen twee syntaxisbomen, mogelijk zijn met behulp van functionele programmeertalen. In dit laatste hoofdstuk worden de resultaten van de creatie van de taaltransformator, die deze taaltransformatie mogelijk maakt, geëvalueerd om te kijken wat dit onderzoek heeft bijgedragen aan het vakgebied. Tot slot worden interessante punten gegeven die tot vervolgonderzoek kunnen leiden.

8.1 Evaluatie

Het onderzoek heeft uit twee delen bestaan die samen een verbetering in het softwareontwikkelingsproces beogen, door bij te dragen aan de oplossing van het communicatieprobleem tussen verschillende domeinexperts. Het eerste deel bestaat uit het modelleren van een domeingebied tot een grammatica en het tweede deel uit het transformeren van grammatica-instanties naar een andere representatie. Het transformeren is op twee manieren mogelijk: tussen tekst en grammatica of tussen twee grammatica’s.

8.1.1 Grammaticacreatie

De grammatica’s voor de taaltransformator zijn in dit onderzoek in ADTs of GADTs beschreven. In beide datatypes is het mogelijk om de volgorde van taalelementen vast te leggen. De types zijn non-terminals en de dataconstructoren kunnen zowel non-terminals als terminals zijn. Dat laatste komt voort uit de aard van de functionele programmeertalen. Immers elke type-instantie, van een ADT of GADT, moet een dataconstructor hebben en zodoende ontstaat bij de verwijzing naar een andere non-terminal een extra dataconstructor. Een voorbeeld van een non-terminal datatype is de dataconstructor `Meer` van het type `Meerdere a`, een voorbeeld van een terminal datatype is de dataconstructor `Als` van het type `Als`.

De keuze tussen ADTs of GADTs heeft te maken met het gebruik van het datatype. Bij een ADT moeten de types van alle argumenten beschreven zijn bij de definitie van het datatype. Bij een GADT daarentegen mogen types opengelaten worden en worden ze door de type-instantie bepaald. De transformatietaal uit Sectie 5.3.2 is niet met ADTs te beschrijven vanwege het veranderende resultaattype van elke transformatiestap. GADTs zijn vooralsnog niet mogelijk in Clean maar in Haskell kunnen op een GADT weer geen generische functies worden gedefinieerd op de wijze waarop generische functies te definiëren zijn in Clean en Haskell. Met andere generische werkwijzen zijn generische functies op GADTs wel mogelijk [25].

In het onderzoek is gekozen om de abstracte en concrete grammatica te definiëren in ADTs omdat op een ADT generische functies kunnen worden gedefinieerd. De transformatietaal is in GADTs gedefinieerd vanwege de veranderende types bij elke transformatiestap.

Het beschrijven van een grammatica kan aan de hand van de regels/grammatica voor ADTs of GADTs. De keuze voor het soort datatype is afhankelijk van het doel van de grammatica. Als de grammatica gebruikt wordt in generische functies dan moet voor ADTs worden gekozen. Voor het interpreteren van de grammatica, zoals de transformatietaal, krijgt een GADT de voorkeur omdat op basis van de instantie het uiteindelijke type wordt bepaald.

8.1.2 Grammaticatransformatie

In dit onderzoek worden twee soorten transformaties onderscheiden. Ten eerste de transformatie tussen een syntaxisboom en tekst en ten tweede de transformatie tussen twee syntaxisbomen.
8.1.2.1 Pretty-printing en parsing

De transformatie tussen een syntaxisboom en tekst is tweezijdig namelijk de transformatie van een syntaxisboom naar tekst (dit wordt pretty-printen genoemd) en de transformatie van tekst naar een syntaxisboom (dit wordt parsing genoemd).

In dit onderzoek is de pretty-print-functie eerst handmatig geschreven; de resultaten neigden naar een generische aanpak en daarvoor is ook gekozen. Voor de dataconstructoren die een terminalwaarde representeren moeten pretty-print-instanties worden gedefinieerd die de terminalwaarde opleveren. Bijvoorbeeld voor leestekens en dataconstructoren die woorden representeren zoals Als en Dan. Hierna kan de functie voor ADTs worden afgeleid en kunnen instanties worden geprint.

De naam pretty-printing heeft het woord “pretty” in zich wat aangeeft dat de uitvoer “mooi” moet zijn, maar dat is zonder extra toevoegingen niet het geval. Een standaard generische printfunctie zal zichzelf recursief aanroepen, alle uitvoer wordt achter elkaar gezet en er wordt geen rekening houden met de opmaak van de tekst. Tekst zonder opmaak is minder goed leesbaar, programmcode wordt bijvoorbeeld duidelijker door het inspringniveau van de regels. Zo is het bij het taalelement IFC duidelijker als de woorden Als en Anders op hetzelfde inspringniveau staan. Dit is niet geval bij een afgeleide generische functie voor het type Stmt, waar IFC deel van uitmaakt.

Om woorden zoals Als en Anders op hetzelfde inspringniveau te printen moet een functie-instantie geschreven worden voor het type Stmt die met het inspringniveau rekening houdt. Wanneer functies zoals deze gedefinieerd zijn kan de uitvoer wel opgemaakt worden. Maar het definiëren van veel functie-instanties past echter niet bij het generisch programmeren: er wordt alsnog specifiek aangegeven wat met welke types moet gebeuren. Daarnaast moeten in Clean nog wat trucs uitgehaald worden om de generische functie generiek te houden. Het is namelijk in Clean niet mogelijk om andere generische functies aan te roepen in een generische functie.

Het generiek specificeren van de pretty-printer ten opzichte van een volledig handgeschreven variant heeft voordelen: alle eenvoudige instanties en constructies kunnen, na eenmalig gedefinieerd te zijn, worden afgeleid. Voor specifieke wensen en eisen is kennis van het gebruikte generische raamwerk nodig om het doel te bereiken.

Naast de generische pretty-printer is voor het herkennen van tekst een generische parser geschreven. Voor de generische parser gelden dezelfde eigenschappen als voor de pretty-printer, alleen van tekst naar een boomstructuur. In plaats van een dataconstructor als tekst te printen wordt op basis van tekst een dataconstructor gecreëerd.

Bij het definiëren van de generische parser wordt nog verder van de generische gedachte afgeweken. De parser zou de inverse van de pretty-printer moeten zijn, dus alle handelingen die voor de pretty-printer zijn gedaan moeten ook voor de generische parser worden gedaan in omgekeerde volgorde. Daarnaast kunnen parsers vaak overweg met extra karakters die er niet toe doen, zoals witruimte. Maar de generische parser parseert precies wat gedefinieerd is en moet ook precies de instanties opleveren die gevraagd zijn, dus ook gedefinieerde witruimte in de concrete grammatica. Als een datatype NewLine geparseerd moet worden dan moet de parser \r\n consumeren. De generische parser kan niet overweg met extra of uitbrekende witruimte, wat het schrijven van invoer voor de parser lastig maakt.

Om extra of ontbrekende karakters te kunnen parseren moet de parser worden uitgerust met extra functie-instanties. Op dit punt wordt de generische wereld losgelaten: elke instantie wordt alsnog specifiek gedefinieerd. Een andere mogelijkheid om extra of ontbrekende karakters te kunnen parseren is het definiëren van een specifieke grammatica voor het parseren waarin bepaalde karakters altijd genegeerd worden. Voor deze aanpak moeten twee grammatica’s worden gedefinieerd: één om te printen en één om te parseren. Ook is een transformatie tussen die grammatica’s nodig.

Het generiek specificeren van een parser voor een grammatica, waar de invoer “los” voor geschreven kan worden met dus extra of ontbrekende witruimte, is minder geschikt. Voor die parser moeten veel specifieke instanties geschreven worden, waardoor het generische raamwerk niet of nauwelijks gebruikt wordt. Voor het specificeren van een parser voor een taal die exact geschreven moet zijn, is een generische parser wel geschikt omdat een generische parser precies parseert wat gedefinieerd is. Voor de AST zelf als interne representatie of ontkoppeling bijvoorbeeld. Extra functionaliteit buiten de grammatica om betekent dat van de generische gedachte wordt afgeweken.
8.1.2.2 Syntaxistransformaties

De tweede transformatievorm in dit onderzoek is de transformatie tussen twee grammatica’s ofwel syntaxistransformatie. Hierin worden grammatica-instanties herschreven naar een andere grammatica. In het herschrijven worden twee varianten onderkend: het decoreren van een grammatica en het herschrijven naar een andere grammatica. In beide gevallen zijn de grammatica’s isomorf.

Voor syntaxistransformaties is een taal ontworpen die elementsgewijs de doelgrammatica kan vullen vanuit een brongrammatica. Deze taal is in een GADT gedefinieerd. De grammatica is zo gedefinieerd dat deze door een evaluatiefunctie ingelezen kan worden en een transformatiefunctie oplevert van een brongrammatica naar een doelgrammatica. De taal is regelgebaseerd en relateert op het hoogste niveau twee types, ofwel grammatica’s, met elkaar. Daaronder moet per dataconstructor een transformatieregel gedeclareerd worden die de broninstantie transformeert naar de doelinstantie.


Met de transformatietaal kunnen grammatica’s eenvoudig gedecoreerd of geconcretiseerd worden. Dit kan omdat de brontaal en doeltaal isomorf zijn en dicht bij elkaar liggen. De decoratie bestaat uit het toevoegen van witruimte, leestekens en woorden. Zo ontstaat een transformatie van een abstracte naar een concrete syntaxis. De transformatie de andere kant op, van een concrete naar een abstracte syntaxis, werkt op dezelfde manier alleen worden dan elementen in de bron genegeerd.

De transformatie tussen verschillende grammatica’s (syntaxistransformatie), bijvoorbeeld tussen de Wettekst-taal en een programmeertaal, wordt gedefinieerd op de abstracte syntaxis van beide talen. Deze talen hebben een overlappend uitdrukkingsgebied en zijn isomorf, maar liggen verder uit elkaar dan bij de decoratie van een grammatica. Een syntaxis transformatie hoeft niet per definitie omkeerbaar te zijn. Een transformatie naar een stuk documentatie is bijvoorbeeld niet omkeerbaar. Syntaxistransformaties zijn ook mogelijk met de transformatietaal, maar lastiger dan de decoratie van een syntaxis. Er is immers geen één-op-één relatie tussen de brongrammatica en de doelgrammatica. Daarom moeten eerst de gewenste taalstructuren in de doeltaal bekend zijn: hoe moeten instanties van de brontaal in de doel-taal eruit komen te zien? Op basis daarvan kan de transformatie geschreven worden.

De transformatietaal transformeert op basis van dataconstructoren en dat maakt het soms lastig om een transformatie te specificeren. Het transformeren van meerdere non-terminals achter elkaar blijft een uitdaging. De transformatiefunctie plaatst na elke transformatieregel een dataconstructor en hierbij moet worden nagedacht welke dataconstructor wanneer wordt geplaatst. Bijvoorbeeld in een transformatie naar het type Expr vanuit een Expressie, waarin worden twee dataconstructoren geplaatst namelijk Expr’ en de dataconstructor voor het huidige element. Of bij de creatie van meerdere elementen bij de dataconstructor Comp, daarbij is het handig om eerst een leeg Stmt` te creëren waarna er altijd een instantie van Stmt komt. Dus eerst een Comp Skip creëren en het volgende argument transformeren naar een Stmt.

De transformatietaal is krachtig genoeg om beide transformatievarianten vast te kunnen leggen en uitvoerbaar te maken. De taal is echter gemodelleerd in een GADT en daardoor kan geen gebruik worden gemaakt van generische functies. Het is dus niet mogelijk om een generische pretty-printer of parser voor de transformatietaal af te leiden.

Het maken van een handgeschreven pretty-printer of parser voor de taal is een tijdrovende bezigheid. Alle functietypes die gebruikt worden moeten worden voorzien van een label. Die labels worden dan geprint en/of geparseerd. Alle gelabelde functies zijn eenvoudige functies die of het n-de argument opleveren of een dataconstructor controleren. Deze functies moeten allemaal met de hand geschreven en gelabeld worden. Daardoor is het schrijven van een pretty-printer en parser meer werk dan het direct uitschrijven van de transformatie.
De conclusie is dat met een bron- en doelgrammatica en een transformatiegrammatica is aangetoond dat verschillende domeinexperts over hetzelfde domein kunnen communiceren in hun eigen jargon. De transformaties maken het mogelijk om de vastgelegde kennis te transformeren naar het desbetreffende domeingebied. Speciale grammatica's en transformaties maken het mogelijk om abstracte kennis duidelijk aan de gebruiker beschikbaar te stellen. Deze kennis kan op zijn beurt weer geparseerd en getransformeerd worden naar een ander domeingebied.

De transformatietaal met bijbehorende taaltransformator uit dit onderzoek maakt de twee transformatievarianten mogelijk. Door gebruik te maken van de taaltransformator met verschillende grammatica's kan het communicatieprobleem worden verkleind.

8.2 Contributie

De contributie van dit onderzoek is de taaltransformator die datatypes kan herschrijven. De transformator breekt datatypes open en bouwt stapsgewijs nieuwe instanties op basis van functiecompositie. Naast de transformator zijn, op basis van de grammatica in ADTs, generische functies gedefinieerd die een ADT-instantie kunnen printen of parseren. In deze scriptie is de types-als-grammatica-aanpak verder geëxpland en op die wijze is een executeerbare transformatietaal ontworpen. Deze taal is beschreven in een GADT en is met bijbehorende evaluatiefuncties uitvoerbaar.

De taaltransformator en grammatica's zijn in twee functionele programmeertalen gebouwd: Clean en Haskell. Het geheel is een proof-of-concept om te laten zien dat een dergelijk concept in functionele programmeertalen kan worden gebouwd.

De methode van dit onderzoek is technisch van aard: de transformaties zijn beschreven met een combinatie van functies en Arrows. Daardoor is het definiëren van transformaties alleen weggelegd voor functioneel programmeurs. Het maken van grammatica's is mogelijk door mensen die enige kennis hebben van het maken van grammatica's, omdat de grammatica's sterk op (E)BNF lijken. Er ontstaat een scheiding tussen grammatica's en transformatie definities.

Echter wanneer de transformatietaal geregistreerd kan worden als tekst en de creatie van de bijbehorende printers en parsers gerealiseerd wordt, dan is het mogelijk om de transformaties in hun eigen taal te definiëren. Het gevolg daarvan is dat de transformatiebouwers alleen de transformatietaal hoeven te kennen. Concluderend, de grammatica's zijn met enige kennis van grammatica's te beschrijven. Voor de transformatie definities daarentegen is kennis nodig van functionele programmeertalen.

8.3 Vervolgonderzoek

In deze scriptie is aangetoond dat met functionele programmeertalen een basis voor een taaltransformator kan worden gelegd. Tijdens de creatie van de grammatica's en de transformator zijn keuzes gemaakt die mogelijk anders hadden gekund en zijn nieuwe ideeën opgedaan. Daarom eindigt deze scriptie met een aantal suggesties voor vervolgonderzoek.

Het eerste punt voor vervolgonderzoek is het toestaan van linksrecursieve grammatica's. In [4] is een methodo toegepast om linksrecursie op parser niveau op te lossen. Door deze methode toe te passen op de types-als-grammatica-aanpak worden de grammatica's eenvoudiger. Het is dan niet nodig om extra types te definiëren die "links" en "rechts" identificeren, zoals de types Expr' (links) en Expr (rechts).

Op dit moment moeten alle transformaties specifiek worden gedefinieerd. De inverse van sommige transformaties is op papier eenvoudig af te leiden, bijvoorbeeld de inverse van de decoratie van een grammatica. In dit onderzoek is de Arrow-klasse gebruikt om de transformatie te definiëren. In [3] heeft Alimarine laten zien dat door middel van de BiArrow-klasse de inverse automatisch afgeleid kan worden. Een interessante vraag die gesteld kan worden is of het mogelijk is om de gebruikte
transformatietaal aan te passen zodat deze gebruik maakt van de BiArrow-klasse en dat daardoor de inverse van een transformatie afgeleid kan worden.

Een andere suggestie heeft betrekking op de transformatie naar een andere grammaticadefinitie. In imperative programmeertalen werkt men vaak met templates [18]. Een template is een vooraf gedefinieerd model met daarin een aantal open plekken. Deze open plekken kunnen door de transformatie worden gevuld. Met deze aanpak kan de transformatie tussen talen die verder uit elkaar liggen vereenvoudigd worden. De transformatie van een domeinspecifieke taal naar een programmeertaal bijvoorbeeld. In de programmeertaal zijn veelal meer grammaticaregels en woorden nodig om dezelfde uitdrukking vast te leggen. In de transformatietaal, zoals die nu is, zijn daardoor veel introductiestappen nodig om alle elementen op de plaats te krijgen. Een template kan het aantal introductiestappen doen verminderen door alleen de relevante introducties open te laten.

In deze scriptie zijn alleen transformaties beschreven die een volledige AST opleveren. Maar het kan zo zijn dat een doeltaal krachtiger of uitgebreider is dan de brontaal. De transformatie kan dan maar een deel van de doeltaal vullen. Is het mogelijk om meerdere deels kennisoverlappende grammatica’s te definiëren die samen naar één volledige AST transformeren? Als dat mogelijk is kunnen grammatica’s extra informatie (detail) voor hun domein bevatten zonder dat het zichtbaar is voor andere domeingebieden. Hierdoor kunnen technische details alleen zichtbaar gemaakt worden voor technisch specialisten.

Tot slot de creatie van een pretty-printer en parser voor de transformatietaal. Dit is nu een tijdrovende klus: alle gebruikte functies moeten gedefinieerd en gelabeld worden en dit resulteert in een typespecifieke pretty-printer of parser. De gebruikte functies zijn allemaal triviaal. Is er een mogelijkheid om deze functies te genereren en te labelen op een manier zodat de pretty-printer en parser deze functies kunnen gebruiken? Als dit mogelijk is kunnen de transformaties in seminatuurlijke taal gedefinieerd worden in plaats van in de technische transformatietaal.
Bibliografie


Appendix A  Parser combinators

:: Parser s r := [s] -> [(r,[s])]

yield :: r -> Parser s r
yield r = \input -> [ ( r, input ) ]
satisfy :: (s -> Bool) -> Parser s s
satisfy p
  = \y ->
    case y of
      [x:xs] ->
      | p x       = [(x,xs)]
      otherwise = []
symbol :: s -> Parser s s | == s
symbol x = satisfy ((==) x)

fail :: Parser r s
fail = \_ -> []

(/
) infixr 6::{Parser s r} (r -> Parser s t) -> Parser s t
(/
) p q
  = \input -> [ ( r2, input2 )
  \\ ( r1, input1 ) <- p input
  , ( r2, input2 ) <- q r1 input1
  ]

// plusP
(\!/) infixr 4 :: (Parser s r) (Parser s r) -> Parser s r
(\!/) pl p2
  = \ input -> case pl input of
    []      = p2 input
    res = res

(\./) infixl 6::{Parser s (r->t)) (Parser s r) -> Parser s t
(\./) p q
  = \ input -> [ (f1 r2, input2 )
    \\ (f1, input1 ) <- p input
    , (r2, input2 ) <- q input1
  ]

(/>\) infixl 6::{Parser s r} (Parser s t) -> Parser s t
(/>\) pl p2 = (\_ y -> y) @$> pl1 /\ .\ p2

(@(>) infix 7 :: (r->t) (Parser s r) -> Parser s t
(@(>) f p = yield f /\ .\ p

(<@>) infixl 7 :: (Parser s r) (r->t) -> Parser s t
(<@>) p f = p /\ yield o f

(/\) infix 6 :: (Parser s r) (Parser s [r]) -> Parser s [r]
(/\) pl p2 = (\r rs -> [r:rs]) @$> pl1 /\ .\ p2

(!@!) :: (Parser s r) -> Parser s [r]
(!@!) parser = (parser /\ (!@!) parser)
  \!/ yield []

(!+@!) :: (Parser s r) -> Parser s [r]
(!+@!) parser = parser /\ (!@!) parser

pDigit :: Parser Char Char
pDigit = satisfy isDigit

pAlpha :: Parser Char Char
pAlpha = satisfy isAlpha

pLetter :: Parser Char Char
pLetter = satisfy isAlphanum
pMatch :: a -> [a] -> [(a,[a])] | == a
pMatch x = satisfy ((==)x)
pWord :: [s] -> Parser s [s] | == s
pWord [] = yield []
pWord [w:ws] = symbol w /:
                    pWord ws
pInt :: ([Char] -> [(Int,[Char])])
pInt = ((symbol '-' /:
pNat) \/
pNat) <@ charList2Int
pNat :: ([Char] -> [([Char],[Char])])
pNat = (!+!) pDigit
pSkipSpace :: (Parser Char a) -> Parser Char a
pSkipSpace p = (!*!) (pWord [' '] \ /
pWord ['	']) ) /\ p
pString x = pWord (fromString x) <@ toString
pToken :: String a -> (Parser Char a)
pToken x y = pWord (fromString x) <@ (\_ -> y)
charList2Int :: [Char] -> Int
charList2Int x = (toInt o toString) x
Appendix B  Transformatie van Prog Var Expr naar ProgC VarC ExprC

\[ u :: a \rightarrow (b ightarrow c) \rightarrow c \]
\[ u \ x \ y \ z = (z \ o \ y) \ x \]

class Transform a b :: a -> b
introduce a b = a

iAls = introduce Als
iDan = introduce Dan
iZolang = introduce Zolang
iDoe = introduce Doe
iAnders = introduce Anders
iNewLine = introduce NewLine
iIndent = introduce Indent
iQuote = introduce Quote
iOpenBracket = introduce OpenBracket
iCloseBracket = introduce CloseBracket
iDot = introduce Dot
iAssignSign = introduce AssignSign
iProgramma = introduce Programma
iBegin = introduce Begin
iEnd = introduce End
iVoerUit = introduce VoerUit
iProc = introduce Proc
iOpenSquareBracket = introduce OpenSquareBracket
iCloseSquareBracket = introduce CloseSquareBracket
iSpaces = introduce (Plus Space)

arrAls x = arr (u x iAls)
arrDan x = arr (u x iDan)
arrZolang x = arr (u x iZolang)
arrDoe x = arr (u x iDoe)
arrAnders x = arr (u x iAnders)
arrNewLine x = arr (u x iNewLine)
arrIndent x = arr (u x iIndent)
arrQuote x = arr (u x iQuote)
arrOpenBracket x = arr (u x iOpenBracket)
arrCloseBracket x = arr (u x iCloseBracket)
arrDot x = arr (u x iDot)
arrAssignSign x = arr (u x iAssignSign)
arrProgramma x = arr (u x iProgramma)
arrBegin x = arr (u x iBegin)
arrEnd x = arr (u x iEnd)
arrVoerUit x = arr (u x iVoerUit)
arrProc x = arr (u x iProc)
arrOpenSquareBracket x = arr (u x iOpenSquareBracket)
arrCloseSquareBracket x = arr (u x iCloseSquareBracket)
arrSpaces x = arr (u x iSpaces)
instance Transform (Prog Var e) (ProgC VarC ec) | Transform e ec
where
  Transform x  = ( (arrProgramma x
                     >>> arrOpenSquareBracket x
                     >>> arr (u x \(Prog o d s) -> ((Transform o)))
                     >>> arrCloseSquareBracket x
                     >>> arrNewLine x
                     >>> arr (u x \(Prog o d s) -> (Transform d))
                     >>> arrNewLine x
                     >>> arrBegin x
                     >>> arrNewLine x
                     >>> arr (u x \(Prog o d s) -> (Transform s))
                     >>> arrNewLine x
                     >>> arrEnd x ) ProgC

instance Transform (Decl v e) (DeclC vc ec) | Transform v vc & Transform e ec
where
  Transform x=:(EmptyDecl) = EmptyDeclC
  Transform x=:(Decl a b c) = ( arrProc x
                               >>> arrSpaces x
                               >>> arr (u x \(Decl a _ _) -> (Transform a))
                               >>> arrSpaces x
                               >>> arrOpenBracket x
                               >>> arr (u x \(Decl _ b _) -> (Transform b))
                               >>> arrCloseBracket x
                               >>> arrNewLine x
                               >>> arrBegin x
                               >>> arrNewLine x
                               >>> arr (u x \(Decl _ _ c) -> (Transform c))
                               >>> arrNewLine x
                               >>> arrEnd x ) DeclC
  Transform x=:(BinDecl a b) = ( arr (u x \(BinDecl a b) -> (Transform a))
                                 >>> arrNewLine x
                                 >>> arr (u x \(BinDecl a b) -> (Transform b))
                                 ) BinDeclC

instance Transform Var VarC
where
  Transform x=:(Var s) = arr (u x \(Var x) -> x) VarC

instance Transform Expr ExprC
where
  Transform x=:(Expr a b c) = ( arr (u x \(Expr a b c) -> (Transform a))
                               >>> arr (u x \(Expr a b c) -> (Transform b))
                               >>> arr (u x \(Expr a b c) -> (Transform c))
                               ) ExprC
  Transform x=:(Expr` e) = arr (u x \(Expr` e) -> (Transform e)) ExprC`

instance Transform Expr` ExprC`
where
  Transform x=:(ExprV v) = arr (u x \(ExprV v) -> (Transform v)) ExprVC
  Transform x=:(ExprS s) = ( arrQuote x
                            >>> arr (u x \(ExprS s) -> s)
                            >>> arrQuote x
                            ) ExprSC
  Transform x=:(ExprB b) = arr (u x \(ExprB b) -> b) ExprBC
  Transform x=:(ExprI i) = arr (u x \(ExprI i) -> i) ExprIC
  Transform x=:(ExprR r) = arr (u x \(ExprR r) -> r) ExprRC
  Transform x=:(BrExpr e) = ( arrOpenBracket x
                             >>> arr (u x \(BrExpr e) -> (Transform e))
                             >>> arrCloseBracket x
                             ) BrExprC
  Transform x=:(Call v e) = ( arrVoerUit x
                             >>> arrSpaces x
                             >>> arr (u x \(Call v e) -> (Transform v))
                             >>> arrOpenBracket x
                             >>> arr (u x \(Call v e) -> (Transform e))
                             >>> arrCloseBracket x
                             ) CallC
instance Transform Opp OppC
where
  Transform x:= (Opp o) = (arr (u x \ (Opp x) -> x)) OppC

instance Transform (Opt a) (Opt b) | Transform a b
where
  Transform Empty = Empty
  Transform (Opt a) = Opt (Transform a)

instance Transform (Stmt v e) (StmtC vc ec) | Transform v vc & Transform e ec
where
  Transform x:= (Comp l r) = (arr (u x \ (Comp a b) -> (Transform a))
    >>> arrDot x
    >>> arrNewLine x
    >>> arr (u x \ (Comp a b) -> (Transform b))
  ) CompC
  Transform x:= (Stmt s) = arr (u x \ (Stmt s) -> (Transform s)) StmtC

instance Transform (StmtC` v e) (StmtC` vc ec) | Transform v vc & Transform e ec
where
  Transform x:= (Assign v e) = (arr (u x \ (Assign v e) -> (Transform v))
    >>> arrSpaces x
    >>> arrAssignSign x
    >>> arrSpaces x
    >>> arr (u x \ (Assign v e) -> (Transform e))
  ) AssignC
  Transform Skip = SkipC
  Transform x:= (If e a b) = (arrAls x
    >>> arrSpaces x
    >>> arr (u x \ (If e a b) -> (Transform e))
    >>> arrSpaces x
    >>> arrDan x
    >>> arrNewLine x
    >>> arr (u x \ (If e a b) -> (Block (Transform a)))
    >>> arrNewLine x
    >>> arrAnders x
    >>> arrNewLine x
    >>> arr (u x \ (If e a b) -> (Block (Transform b)))
  ) IfC
  Transform x:= (While e s) = (arrZolang x
    >>> arrSpaces x
    >>> arr (u x \ (While e s) -> (Transform e))
    >>> arrSpaces x
    >>> arrDoe x
    >>> arrNewLine x
    >>> arr (u x \ (While e s) -> (Block (Transform s)))
  ) WhileC
Appendix C  Hulpparsers transformatietaal

-- Label TransLangC
parse_TransLangC_l f i = pWord "Transformatie: "
  "/\_ -> pString" 
  "/\_ -> pWord " :==\r
" 
  "/\_ -> f (i++"t")" 
  "/\_ -> yield (Label s x)"

-- TransLangC
parse_TransRuleC f g c i = pWord i 
  "/\_ -> pWord "Als invoer voldoet aan " 
  "/\_ -> pLabel f 
  "/\_ -> g (1++"t")" 
  "/\_ -> c i 
  "/\_ -> z -> yield (TransRuleC (p,r) z)"
parse_EndRuleC i = pWord i 
  "/\_ -> pWord "Einde." 
  "/\_ -> yield EndRuleC"

-- TransRuleC
parse_PutCtorC f g i = pWord i 
  "/\_ -> pWord "creeer een " 
  "/\_ -> pLabel f 
  "/\_ -> g 
  "/\_ -> yield (PutCtorC l c)"
parse_PutCtorEC f i = pWord i 
  "/\_ -> pWord "creeer een " 
  "/\_ -> yield (PutCtorEC l)"

-- ArrLangC
parse_TransIntrC f i = pWord i 
  "/\_ -> pLabel f 
  "/\_ -> x -> yield (TransIntrC x)"
parse_TransArrowC f g i = pWord i 
  "/\_ -> pWord "plaats: " 
  "/\_ -> f i 
  "/\_ -> g i 
  "/\_ -> y -> yield (TransArrowC x y)"
parse_TransArgC f g i = pWord i 
  "/\_ -> pWord "transformeer het resultaat van " 
  "/\_ -> pLabel f 
  "/\_ -> pLabel g 
  "/\_ -> yield (TransArgC p 1)"
parse_TransArgPC f g h i = pWord i 
  "/\_ -> pWord "transformeer het resultaat van " 
  "/\_ -> pLabel f 
  "/\_ -> pLabel g 
  "/\_ -> pLabel h 
  "/\_ -> m -> yield (TransArgPC p i m)"
Appendix D  Cheetah taaldocumentatie

Beschrijving van businessconcept Beslissen Zorg van 2009-01
2.5 partner in huishouden

De deelname met partnerrol is de huishouden.deelname met huishouden.deelname.deelnamerol.Partner
Indien gevonden
   De partner in huishouden is deelname met partnerrol.burger
Indien niet gevonden
   Indien geval niet uniek
   Einde declaratie

Beschrijving van businessconcept Samenstellen Huishouden Zorgmiss van 2009-01
2.8 beëindigen huishouden

Het te beëindigen huishouden is het Huishouden met Huishouden.aanvrager gelijk aan burger
Indien gevonden
   Pas toe attenderen achterblijvende partner bij te beëindigen huishouden
   Pas toe beëindig geldigheid huishouden bij te beëindigen huishouden
Indien niet gevonden
   Indien geval niet uniek
   Einde declaratie
Appendix E  Tussenresultaten transformatie Wetteksten_{AST} naar While_{CST}

Declaratie n De Space (Identifier n "relevanteBesturing") Space (Selectie n Is Space De Space (Identifier n "besturingBetaalAdviezen") Space Met Space (Vergelijking n (Verwijzing n (Identifier n "BesturingBetaalAdviezenToeslagregeling"))) Space Is Space (Verwijzing n (Identifier n "Regeling"))) NewLine Indien Space Gevonden (Blok NewLine (Waarde n (Meer n (Toekenning n De Space (Identifier n "laatsteVrijgavejaar") Space Is Space (Verwijzing n (Identifier n "relevanteBesturingVrijgavejaar"))) NewLine (Meer n (Toekenning n De Space (Identifier n "laatsteVrijgagemaand") Space Is Space (Verwijzing n (Identifier n "relevanteBesturingVrijgagemaand"))) NewLine (Enkel n (Toekenning n De Space (Identifier n "datumAfgifteVrijgave")) Space Is Space (Verwijzing n (Identifier n "relevanteBesturingDatumAfgifteVrijgave"))) NewLine) Indien Space Niet Space Gevonden (Blok NewLine (Waarde n (Meer n (Toekenning n De Space (Identifier n "laatsteVrijgavejaar") Space Is Space (Verwijzing n (Identifier n "tweeDuizendVijf"))) NewLine (Meer n (Toekenning n De Space (Identifier n "laatsteVrijgagemaand") Space Is Space (Verwijzing n (Identifier n "een")) NewLine (Enkel n (Toekenning n De Space (Identifier n "datumAfgifteVrijgave")) Space Is Space (Verwijzing n (Identifier n "eersteDagVanHetJaarVanHetLaatsteVrijgavejaar"))))) NewLine) Indien Space Niet Space Uniek (Blok NewLine Leeg n NewLine) Einde Space Declaratie

Codevoorbeeld 8-1 – Definitie uit Codevoorbeeld 6-17 in Wetteksten_{AST}

Statement (Identifier "relevanteBesturing") (Selectie (Identifier "besturingBetaalAdviezen") (Vergelijking (Verwijzing (Identifier "BesturingBetaalAdviezenToeslagregeling")) (Verwijzing (Identifier "Regeling")))) (Waarde (Meer (Toekenning (Identifier "laatsteVrijgavejaar")) (Verwijzing (Identifier "relevanteBesturingVrijgavejaar")))) (Waarde (Meer (Toekenning (Identifier "laatsteVrijgagemaand") (Verwijzing (Identifier "relevanteBesturingVrijgagemaand")))) (Enkel (Toekenning (Identifier "datumAfgifteVrijgave")) (Verwijzing (Identifier "relevanteBesturingDatumAfgifteVrijgave"))))) (Waarde (Meer (Toekenning (Identifier "laatsteVrijgavejaar")) (Verwijzing (Identifier "tweeDuizendVijf")))(Meer (Toekenning (Identifier "laatsteVrijgagemaand")) (Verwijzing (Identifier "een")))(Enkel (Toekenning (Identifier "datumAfgifteVrijgave")) (Verwijzing (Identifier "eersteDagVanHetJaarVanHetLaatsteVrijgavejaar"))))) Leeg

Codevoorbeeld 8-2 – Definitie uit Codevoorbeeld 6-17 in Wetteksten_{AST}

Decl (Var "relevanteBesturing") (Var "besturingBetaalAdviezen") (Comp (Assign (Var "lQuery") (Expr' (Call (Var "Query.Create") (Expr' (ExprV (Var "besturingBetaalAdviezen"))))) (Comp (Assign (Var "lResult") (Expr' (Call (Var "lQuery") (ExprV (Var "besturingBetaalAdviezen"))))) (Opp (==) (ExprV (Var "Regeling")))) (Stmt (If (Expr (ExprV (Var "lResult.Size()") (Opp (==) (Expr' (ExprI 1))))) (Comp Skip (Comp (Assign (Var "laatsteVrijgavejaar") (Expr' (ExprV (Var "relevanteBesturingVrijgavejaar"))))) (Comp (Assign (Var "laatsteVrijgagemaand") (Expr' (ExprV (Var "relevanteBesturingVrijgagemaand"))))) (Stmt (If (Expr (ExprV (Var "lResult.Size()") (Opp (==) (ExprV (Var "tweeDuizendVijf"))))) (Comp Skip (Comp (Assign (Var "datumAfgifteVrijgave") (Expr' (ExprV (Var "datumAfgifteVrijgave"))))) }}})))) (Stmt Skip))))

Codevoorbeeld 8-3 – Definitie uit Codevoorbeeld 6-17 in While_{AST}
```c
(DeclC Proc (Plus Space) (VarC "relevanteBesturing") (Plus Space) OpenBracket (VarC "besturingBetaalAdviezen") CloseBracket NewLine Begin NewLine (CompC (AssignC (VarC "lQuery") (Plus Space) AssignSign (Plus Space) (ExprC` (CallC VoerUit (Plus Space) (VarC "Query.Create") OpenBracket (ExprVC (VarC "besturingBetaalAdviezen")) CloseBracket))) Dot NewLine (CompC (AssignC (VarC "lResult") (Plus Space) AssignSign (Plus Space) (ExprC` (CallC VoerUit (VarC "lQuery") OpenBracket (ExprC (ExprVC (VarC "BesturingBetaalAdviezenToeslagregeling")) (OppC "==") (ExprC` (ExprVC (VarC "Regeling"))))))) CloseBracket)) Dot NewLine (StmtC(IFC Als (Plus Space) (ExprC (ExprVC (VarC "lResult.Size()")) (OppC "==") (ExprC` (ExprIC 1)))) (Plus Space) Dan NewLine (Block (CompC SkipC Dot NewLine (CompC (AssignC (VarC "laatsteVrijgavejaar") (Plus Space) AssignSign (Plus Space) (ExprC` (ExprVC (VarC "relevanteBesturingVrijgavejaar")))))) Dot NewLine (StmtC (AssignC (VarC "laatsteVrijgagemaand") (Plus Space) AssignSign (Plus Space) (ExprC` (ExprVC (VarC "relevanteBesturingVrijgagemaand")))))) Dot NewLine (StmtC (AssignC (VarC "datumAfgifteVrijgave") (Plus Space) AssignSign (Plus Space) (ExprC` (ExprVC (VarC "relevanteBesturingDatumAfgifteVrijgave")))))))) NewLine Anders NewLine (Block (StmtC (IFC Als (Plus Space) (ExprC (ExprVC (VarC "lResult.Size()")) (OppC "==") (ExprC` (ExprIC 0)))) (Plus Space) Dan NewLine (Block (CompC SkipC Dot NewLine (CompC (AssignC (VarC "laatsteVrijgavejaar") (Plus Space) AssignSign (Plus Space) (ExprC` (ExprVC (VarC "tweeduizendVijf"))))))) Dot NewLine (CompC (AssignC (VarC "laatsteVrijgagemaand") (Plus Space) AssignSign (Plus Space) (ExprC` (ExprVC (VarC "een")))))) Dot NewLine (StmtC (AssignC (VarC "datumAfgifteVrijgave") (Plus Space) AssignSign (Plus Space) (ExprC` (ExprVC (VarC "eersteDagVanHetJaarVanHetLaatsteVrijgavejaar"))))))) NewLine Anders NewLine (Block (StmtC (SkipC) NewLine End (Plus Space) Als NewLine))) NewLine End (Plus Space) Als NewLine))))
```

**Codevoorbeeld 8-4 – Definitie uit Codevoorbeeld 6-17 in WhileCST**