Dankwoord

Alvorens te starten met de eigenlijk thesistekst, wil ik eerst een paar mensen bedanken die een grote steun waren in het maken van deze thesis. In de eerste plaats mijn promotor, Prof. Dr. Jan Van den Bussche, waarmee ik toch een heel aantal namiddagen intenstief heb samengewerkt en ook steeds terecht kon wanneer ik vast zat met mijn thesis. Zeker toen ik het onderwerp even niet meer zag zitten, motiveerde hij mij terug op een manier zodat ik weer met volle moed door kon gaan. Maar ook dienen een aantal andere mensen vermeld te worden. Mijn ouders en zussen die mij vier jaar lang gesteund hebben, net als mijn medestudenten Ward, Vanessa, Annelies, Erwin, ... en alle anderen die hier een vermelding verdienen, maar die door vergetelheid zijn overgeslagen.

Bedankt!

Wim Janssen
Juli 2005

Samenvatting

The eXtensible Stylesheet Language Family (XSL) is een verzameling talen om een XML-document te transformeren naar een nieuwe XML-boom. De inputboom is bijvoorbeeld een database in XML en het resultaat is een presenteerbaar formaat. Je kan vanuit éénzelfde XML-document zowel een brochure in PDF opstellen, als een site in HTML. XSL is opgedeeld in twee deelspecificaties. Als eerste is er de eXtensible Stylesheet Language for Transformations (XSLT) die een gegeven boom omzet in een andere boom. Wanneer deze stap voltooid is, gaan we het bekomen resultaat omzetten in leesbaar formaat. Teksten kunnen ingekleurd worden en verschillende opmaakstijlen zijn voorhande. Dit deel wordt verricht door de eXtensible Stylesheet Language - Formatting Objects (XSL-FO).

In deze thesis gaan we dieper in op XSLT. We willen XSLT in een abstracte syntax gieten en de semantiek formeel beschrijven. Met deze formalisatie gaan we dan op zoek naar allerhande eigenschappen voor XSLT 1.0. Wanneer we daarmee klaar zijn, kijken we naar een essentieel feature dat toegevoegd werd in XSLT 2.0, namelijk de temporary trees.

Na een inleidend hoofdstuk bespreken we boomautomaten. Deze automaten zijn in staat bomen van een bepaalde boomtaal te accepteren of bomen te herschrijven. Natuurlijk leunt dat laatste sterk aan bij XSLT. Ook geeft het de typische onderdelen weer van formele semantiek, zoals het definiëren van configuraties en het opstellen van een transitiefunctie tussen deze configuraties.

Vervolgens gaan we herschrijfsystemen onder de loep nemen. Uiteraard is XSLT een herschrijfsysteem voor bomen. Daarom gaan we in het algemeen enkele bekende eigenschappen van herschrijfsystemen bekijken en definiëren we ook het concept 'confluentie'.

Met deze twee voorgaande hoofdstukken hebben we een stevige basis opgebouwd om een formeel model voor XSLT uit te werken. We beschouwen XSLT als een subtree replacement system, waarop we dus stellingen en lemma's uit het vorige hoofdstuk kunnen toepassen. We trachten aan te tonen dat XSLT confluent is.

Een eerste belangrijke toepassing van ons uitgedokterd herschrijfsysteem is de terminatie van XSLT 1.0. We stellen ons de vraag of het mogelijk is om, voor een gegeven programma en een gegeven inputboom, te achterhalen of de transformatie stopt.

Hiermee eindigen we de studie van XSLT 1.0 en gaan we door met XSLT 2.0. Na een kort overzicht geschetst te hebben van de nieuwe features van

XSLT 2.0, richten we ons op de meest belangrijke nieuwigheid, namelijk de temporary trees. We eindigen de thesis met te kijken welke enorme invloed deze hebben op het aantal berekenbare boomtransformaties die uitgevoerd kunnen worden met behulp van XSLT.

Inhoudsopgave

Lijst van figuren

Hoofdstuk 1

Inleiding

De Extensible Markup Language (XML) [YBP+04] is een semi-gestructureerd dataformaat dat tien jaar na zijn uitvindig nog steeds een enorme groei kent. Steeds meer data wordt in XML-opgeslagen. Het kan hier gaan van relationele databases die geconverteerd worden, tot beschrijvingen van grafische user interfaces. Maar als men naar XML zelf kijkt, doet dat eigenlijk niks. XML is niets anders dan een beschrijving hoe gestructureerde data opgeslagen kan worden. Deze structuur is altijd een boom, waarbij men de start en het einde van knopen aangeeft met zogenaamde tags. Het zijn de allerhande bijkomende specificaties die XML krachtig maken. Om te beginnen heb je een heel arsenaal aan mogelijkheden om te specificieren hoe een XML-bestand in elkaar moet zitten. Als eerste was er de Document Type Definition (DTD). Deze beschreef met behulp van een grammatica de ouder-kind-relatie van een XML-boom. Omdat DTD's niet echt krachtig zijn, volgden er een hele reeks andere methodes om de structuur van een XML-boom vast te leggen. Zo is er ondertussen XML-Schema, Relax NG, Schematron, DSD, DTD++, ... Buiten deze schematalen zijn er ook nog de querytalen voor XML. Hieronder hebben we XPath, XPointer, Xquery, ...

Een belangrijk kenmerk van XML is de scheiding van de data en de opmaak van de data. Er is dan ook een aparte specificatie die vertelt hoe we de opmaak van een XML-bestand kunnen genereren, de eXtensible Stylesheet Language Family (XSL). Deze bestaat eigenlijk uit twee delen. Als eerste is er de eXtensible Stylesheet Language Transformations (XSLT). Deze gaat een XML-boom omzetten in een nieuwe XML-boom. Dan komt het tweede deel van XSL, de eXtensible Stylesheet Language Formatting Objects (XSL-FO). Hierin wordt bepaald hoe de nieuwe getransformeerde boom getoond moet worden aan de gebruiker. Men kan allerhande opmaak kiezen voor de data.

In dit inleidend hoofdstuk gaan we dieper in op XSL. Het nut van trans-

formaties wordt besproken en het onstaan van XML en XSLT geschetst. Daarna wordt de werking uitgelegd aan de hand van een voorbeeld.

1.1 Waarom transformeren?

Er zijn twee belangrijke redenen waarom we een XML-bestand willen transformeren. Een eerste reden is het versturen van data tussen twee applicaties. Wanneer de eerste applicatie data genereert in een bepaald formaat, kan het zijn dat de tweede applicatie deze informatie niet kan gebruiken. De data moet in een ander formaat worden gepresenteerd aan de applicatie. Door een simpele XML-transformatie is de tweede applicatie misschien wel in staat de informatie te begrijpen. Een dergelijke transformatie is waarschijnlijk veel eenvoudiger op te stellen dan de applicatie aan te passen aan ieder mogelijk input-formaat.

Een tweede reden is dat we onze data willen scheiden van de representatie. Stel dat er een XML-bestand bestaat met daarin gegevens over producten die een winkel verkoopt. Wanneer een online-catalogus beschikbaar gesteld wordt, kan deze gegenereerd worden door een transformatie van het XML-bestand naar een HTML-bestand. Als er ook een papieren versie van de catalogus moet bestaan, wordt er misschien gekozen voor een PDF-bestand. Een nieuwe transformatie vanuit hetzelfde XML-bestand kan deze catalogus maken. Op deze manier houden we onze data maar op één plaats bij, namelijk in het XML-bestand. Afhankelijk van het gebruikte doel, genereren we verschillende presentaties van deze data.

1.2 Het onstaan van XSLT

Op het World Wide Web is HTML de taal om een website op te stellen. In HTML wordt gespecificeerd wat de inhoud is van de site (welke informatie er getoond wordt), samen met de opmaak voor die data. De data staat binnen speciale tags, zodat een browser weet aan de hand van deze tags hoe hij bepaalde tekst moet tonen. Oorspronkelijk waren er maar weinig opmaakmogelijkheden. Toch wilde gebruikers van HTML ook tabellen, kleuren, ... gebruiken om een site te verfraaien. Hiervoor moesten dan weer nieuwe tags gevonden worden. Daardoor kreeg men niet alleen problemen met browsers die deze nieuwe tags nog niet konden interpreteren, maar ook dat HTML niet langer een eenvoudige taal bleek te zijn.

De oplossing voor deze problemen is het scheiden van de data en zijn opmaak. Deze manier van werken was al langer bekend. In de jaren '60-'70

werkte men reeds met de Standard Generalized Markup Language (SGML). SGML laat heel veel vrijheid toe, waardoor de taal eigenlijk moeilijk te begrijpen en te ontleden was. Dit gaf aanleiding tot XML. XML is een subset van SGML, waarbij men zich strikter aan bepaalde regels moest houden. Zo was in SGML een sluittag niet altijd verplicht, terwijl dat bij XML wel zo is. Een klein voorbeeldje hiervan is de
-tag in HTML. Volgens de XML-specificatie is
 op zich ongeldig omdat de sluittag ontbreekt.
 of
</br> wordt wel goedgekeurd.

De eerste technieken om SGML om te zetten naar een bepaald (meestal typografisch) formaat, waren zeer specifieke en vaak dure tools. Begin van de jaren '90 kwam echter de Document Style Semantics and Specification Language aanzetten (DSSSL). Het idee was om een standaard te ontwikkelen om SGML te converteren naar een presentatieformaat. Een belangrijke paper in dit verband was één van C.M. Sperberg-McQueen en Robert F.Goldstein [SG95]. Hierin vermeldden ze hoe een SGML bestand zou geïnterpreteerd kunnen worden door browsers. HTML is maar een mogelijk SGML-bestand. Voor HTML kennen de browsers de betekenis van de verschillende tags. Een voorbeeldje hiervan is dat HTML weet dat de tekst binnen tags met naam 'b' in het vetjes (bold) getoond moet worden. Deze betekenis is hard-coded opgeslagen in de code van een browser. Het idee is nu om een brower ook willekeurig SGML-bestand te laten tonen. Hierbij wordt dan voor dat SGML-bestand een stylesheet bijgehouden. Deze stylesheet bepaalt hoe de opmaak moet zijn van tekst die binnen een bepaalde tag van het SGML-bestand staat.

In 1995 werd er door het World Wide Web Consortium (W3C)[Ber] een workshop over stylesheets gehouden in Parijs. Hierin werden een aantal wenselijke eigenschappen van DSSSL toegelicht. James Clark was één van de sprekers over DSSSL. Hij vertelde dat DSSSL uit twee delen bestond, namelijk een transformatietaal en een opmaaktaal. Het moest mogelijk zijn in een stylesheet andere stylesheets op te roepen, zodat er eventueel bestaande opmaakregels overschreven konden worden, net zoals in Cascading Style Sheets (CSS). Hij vermeldde zo een reeks features voor DSSSL.

Op 21 augustus 1997 kwam er een eerste voorstel van een nieuwe stylesheettaal: the eXtensible Stylesheet Language Family, kortweg XSL. Deze was vooral gericht op XML in plaats van SGML. De syntax van de taal zelf was ook XML. XSL was eigenlijk een deel van DSSSL, hoewel het ook uitbreidingen bevatte die in DSSSL niet mogelijk waren. De auteurs stelden dan ook voor om DSSSL met dezelfde nieuwe mogelijkheden uit te breiden. Net zoals James Clark reeds aangaf, werd XSL opgesplitst in twee delen. Enerzijds is er het transformatiegedeelte XSLT en anderzijds het gedeelte dat de opmaak bepaalt, XSL-FO (Formatting Objects). Op 16 november 1999 kwam de eer-

ste afgewerkte specificatie, XSLT 1.0 [Cla99]. Ondertussen is men, na XSLT 1.1 [Cla01], bezig met de ontwikkeling van XSLT 2.0 [Kay05].

1.3 De werking van XSLT

XSLT is een rule-based taal, wat wil zeggen dat een programma bestaat uit een aantal regels. Ze verklaren elk hoe een bepaald onderdeel verwerkt dient te worden. Deze regels, ook wel de templates genoemd, moeten niet in een bepaalde volgorde staan, ze hebben dus geen invloed op de volgorde van output. Dit is wat XSLT een declaratieve taal maakt, omdat je specificeert wat de output moet zijn wanneer bepaalde patronen voorkomen. In een procedurele taal (zoals C) vermeld je precies welke taken in welke volgorde uitgevoerd moeten worden.

Tijdens de uitvoering van een XSLT-stylesheet wordt er steeds een tupel met vier belangrijke parameters bijgehouden. Deze zogenaamde context bevat:

- één contextnode: deze vertelt op welke positie we zijn binnen het inputdocument,
- een variabeletoekenning: voor iedere variabele, die zichtbaar is in de huidige plaats van de stylesheet, wordt de waarde bijgehouden,
- een contextsize: de lengte van de contextlist. Dit is een verzameling knopen uit het inputbestand die we allemaal dienen te verwerken. De contextlist bevat dus zeker de contextnode. De inhoud van deze lijst is bijvoorbeeld de verzameling knopen die we moeten verwerken om een volledige 'for-each' uit te voeren.
- een contextpositie: de plaats van de contextnode in de contextlist.

Wanneer een transformatie start, wordt als context altijd het tupel genomen bestaande uit een lege variabeletoekenning (omdat er nog geen variabelen zijn) en als contextsize en contextpositie de waarde 1. Als contextnode wordt de zogenaamde 'virtuele root' van het XML-bestand genomen. Deze 'virtuele root' is een belangrijk principe in XSLT, wat we zullen uitleggen aan de hand van een voorbeeldje. Beschouw even het XML-bestand uit figuur 1.1. Met de 'root van het XML-bestand' wordt de knoop met label 'message' bedoeld. De 'virtuele root' is een denkbeeldige knoop met als kinderen de programming instruction die aangeeft dat we met XML versie 1.0 werken, de daaropvolgende commentaarlijn en tenslotte de knoop met label 'message'.

Dit is gedaan zodat men, indien gewenst, ook deze extra regels kan achterhalen vanuit XSLT. Ook is het dan mogelijk om met XML-bestanden te werken die meerdere XML-bomen bevatten. In het vervolg zullen we, tenzij expliciet vermeld, echter met 'root' steeds de wortel van de XML-boom bedoelen, dus niet de virtuele root.

XSLT werkt samen met XPath [CD99], een patroontaal voor het selecteren van knopen uit een XML-boom, hier dus het inputbestand. Een XPath-expressie wordt altijd geëvalueerd vanuit de context.

De variabelen in XSLT kunnen best vergeleken worden met final variabelen in Java. Éénmaal ze een waarde hebben gekregen, kan deze waarde niet meer veranderen. Hoewel het type niet vermeld kan worden, zijn er vijf mogelijke types:

- Node-set: een verzameling knopen van de inputboom,
- Number: een willekeurig getal,
- Boolean: de booleaanse waarde true of false,
- String: een willekeurige string,
- Result-tree-fragment.

Deze laatste zorgde voor heel wat verwarring in het begin van XSLT. Een result-tree-fragment is een boom waarop men enkel string-operaties mag uitvoeren. Het is dus in het bijzonder niet toegestaan om XPath-expressies los te laten op een dergelijke variabele. Het kan dus eigenlijk alleen maar gebruikt worden om vaak terugkerende fragmenten in de output als variabele op te slaan. Wanneer men een variabele instantieert met het resultaat van een template, dan zal deze altijd het type result-tree-fragment hebben. Dit heeft als nadeel dat het resultaat van een uitwerking van een template niet meer verwerkt kan worden als een boom. Een variabele kan ook voorkomen als parameter bij het oproepen van een template.

We zullen nu aan de hand van een aantal voorbeelden de werking en de verschillende onderdelen van XSLT uitleggen.

1.3.1 Voorbeelden

Hello World

Sinds het boek 'The C Programming Language' van Brian W. Kernighan en Dennis M. Ritchie [KR88] is het de traditie dat het eerste voorbeeldprogramma dat gegeven wordt 'hello world' uitschrijft. In figuur 1.1 staat een XML-bestandje, samen met een stylesheet.

```
  <?xml version="1.0" encoding="UTF-8"?>

<!-- Eerste voorbeeld-programma-->
<message>
    Hello World!
</message>
```

```
  <?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:template match="/">
        <html>
            <xsl:copy-of select="./message/text()"/>
        </html>
    </xsl:template>

</xsl:stylesheet>
```

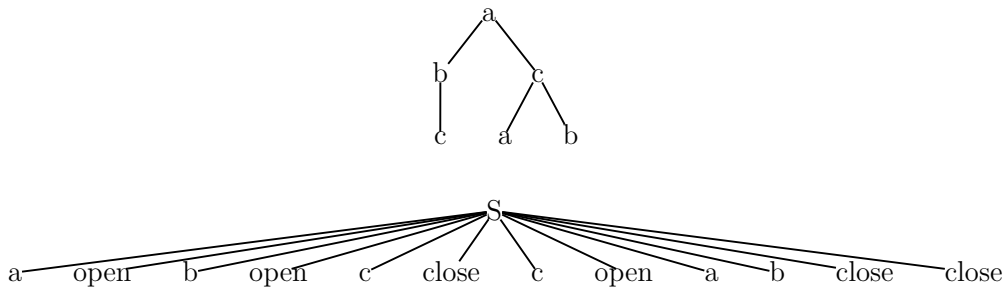
Figuur 1.1: Hello world

De eerste regel in de stylesheet geeft aan dat we met XML versie 1.0 werken, net zoals in ieder ander XML-bestand (merk op dat XSLT zelf ook in XML geschreven is). Op de tweede regel begint de eigenlijke stylesheet. We gebruiken versie 1.0 van XSLT en de namespace van XSLT noemen we 'xsl'. Deze twee regels zullen bijna altijd letterlijk herhaald worden in iedere stylesheet. Hoe gebeurt nu de transformatie?

De eerste taak van een XSLT-transformatie is het zoeken naar een rule waarvan het match-attribuut de virtuele root selecteert. Merk op dat deze XPath-expressie geëvalueerd wordt vanuit de virtuele root. In dit XSLT-programma is er maar één rule. Het match-attribuut heeft als waarde "/ ". Deze selecteert zeker de virtuele root. We kijken dan naar de template binnen de gevonden rule. Deze maakt eerst een knoop gelabeld `html`. Het kind van deze knoop in de output is het resultaat van het kind van deze knoop in de

```
  <html>
    Hello World!
</html>
```

Figuur 1.2: Hello world, resultaat



Figuur 1.3: Een diepe en een brede boom

stylesheet. Dit is een nieuw statement, namelijk 'copy-of'. Deze gaat het resultaat van de XPath-expressie vermeld in zijn select-attribuut kopiëren naar de output. Wanneer dit gedaan is, is de template volledig verwerkt en stopt het programma. Het resultaat is de HTML-pagina van figuur 1.2.

Van diepe naar brede boom

Het volgend voorbeeld laat een iets ingewikkeldere boomtransformatie zien: het idee is om een geneste boomstructuur om te vormen naar een brede boomvoorstelling. In plaats van kinderen onder een knoop te hangen, gebruiken we haakjes om de nesting voor te stellen (omdat de symbolen '(' en ')' geen geldige XML-tagnamen zijn, zullen we deze voorstellen met 'open' en 'close'). Figuur 1.3 geeft dit visueel weer.

We nemen aan dat iedere tag van de XML-boom uit een eindig alfabet is gekozen. De inputbomen moeten bijvoorbeeld voldoen aan een DTD, die enkel de tagnamen a, b en c definieert. Figuur 1.4 laat een mogelijk XML-inputbestand zien, waarbij het tag-alfabet bestaat uit de symbolen a, b en c. We willen deze boom nu omvormen zoals in figuur 1.3. Het XSLT-programma dat deze bewerking uitvoert, is beschreven in figuur 1.5. We gaan dit nu stap voor stap bespreken. Het XSLT-programma begint, zoals altijd, met het zoeken naar een template die in zijn match-attribuut een XPath-expressie

```
  <?xml version="1.0" encoding="utf-8"?>

<a>
    <b>
        <c>
    </b>
    <c>
        <a/>
        <b/>
    </c>
</a>
```

Figuur 1.4: Diepe boom als input

heeft staan die de virtuele root van het inputbestand selecteert. In deze stylesheet zijn er drie mogelijke templates, waarvan geen enkele die root selecteert. We gaan dan als oplossing op zoek naar de regel die de root van het XML-bestand selecteert. De eerste template selecteert alle knopen, dus ook zeker die root. We starten dus met de uitvoer van die template, waarbij de contextnode de root is. Als eerste wordt nu een knoop S aangemaakt. Als kind van deze S komt het resultaat van het apply-templatesstatement. Dit commando gaat op zoek naar een template binnen de stylesheet waarvan het match-attribuut, geëvalueerd vanuit de huidige context, de knoop selecteert die vermeld is in zijn select-attribuut. Ook wordt er geëist dat de gevonden template als mode 'makeBroadRepresentation' heeft. Er kan zo maar één gevonden worden, namelijk de tweede template van de stylesheet. Hier gaan we dus verder. De contextknoop verandert nu in de knoop die geselecteerd werd door de apply-templates. Het eerste commando dat we tegenkomen is een choose. Deze gaat in zijn kinderen op zoek naar de eerste test die voldoet. Voldoen komt in dit geval overeen met het feit dat de evaluatie van de test vanuit de contextknoop een niet-lege verzameling knopen oplevert. Kan er zo geen test gevonden worden, dan gaan we verder met het otherwise-kind. De enige test die er staat vermeld, voldoet in ons geval. De contextknoop is trouwens de knoop a en de test vraagt of er kinderen zijn. We voeren dan de code uit die onder de test vermeld staat. Als eerste wordt een template opgeroepen met de naam 'makeNode'. We springen dus naar de eerste regel van die template zonder de contextknoop te veranderen. Opnieuw komen we een choose tegen. Deze zal een knoop maken met dezelfde tag als die van de

huidige contextknoop (hier maken we volop gebruik van het feit dat de tags uit een eindig alfabet komen). Wanneer dit gedaan is, gaan we terug verder met de eerste instructie na de call-template. We maken dus een blad met tag 'open' aan en vervolgens komt er een nieuwe apply-templates, waarna een 'close'-knoop gemaakt wordt. De apply-templates werkt hetzelfde als voorheen, maar er is echter een belangrijk verschil. De XPath-expressie in het select-attribuut selecteert nu meer dan één knoop. We gaan dan voor elk van deze knopen apart op zoek naar een template en werken die verder uit, met als contextnode de geselecteerde knoop. De contextsize is het aantal knopen dat geselecteerd werd door het select-attribuut. Bij de volgorde van uitwerken houden we de volgorde van het XML-inputbestand aan. Hierin worden de knopen namelijk volgens pre-order gesorteerd. Als eindresultaat krijgen we het XML-bestand uit figuur 1.6.

Een voorbeeld van for-each

Het vorige voorbeeld maakte gebruik van drie belangrijke statements, namelijk 'choose', 'apply-templates' en 'call-template'. Er zijn nog enkele interessante, waarvan de naam waarschijnlijk al laat vermoeden waarvoor ze dienen: 'if-then', 'for-each' en 'variable'. In dit voorbeeld zullen we deze toelichten. Het idee is om ieder tweede kind van een b knoop in de output te schrijven, op voorwaarde dat dat b-kind in totaal drie kinderen heeft. De waardes 2 en 3 zullen we bijhouden in variabelen. In figuur 1.7 staat de stylesheet, samen met een mogelijk input-XML-bestand.

De uitvoering van de stylesheet begint met het initialiseren van de twee globale variabelen 'aantal' en 'pos'. Globale variabelen worden gedeclareerd buiten alle templates en hun waarde is in heel de stylesheet bekend. Hierna wordt gezocht naar de template die de virtuele root van het inputbestand selecteert. De stylesheet bevat één template en deze selecteert de virtuele root. Hierna volgt de for-each. Deze zal een lijst maken van alle knopen die geselecteerd worden door de XPath-expressie vermeld in zijn select-attribuut. Vervolgens zal de template onder de for-each uitgevoerd worden voor ieder van deze knopen. Als contextnode wordt de geselecteerde knoop gebruikt, als variabeletoekenning de variabeletoekenning die er was voor de aanroep van de for-each. De contextsize is het aantal geselecteerde knopen en de context-position is de plaats van de huidige knoop in die lijst. Voor iedere knoop moet een if-test geëvalueerd worden. De semantiek daarvan is waarschijnlijk duidelijk. De template onder de if-test zal worden uitgevoerd indien de test tot true evalueert. Tot true evalueren kan in een XPath-expressie verschillende betekenissen hebben. Enerzijds kan de test een booleaanse XPath-expressie zijn, anderzijds kan deze expressie ook een node-set selecteren. Er wordt

dan aangenomen dat het resultaat van de test true is als en slechts als deze verzameling knopen niet leeg is.

1.3.2 XSLT-processors

Wanneer je een XSLT-stylesheet hebt gecreëerd, moet deze natuurlijk nog uitgevoerd worden op een XML-bestand. Dit gebeurt door zogenaamde XSLT-processoren. De meest bekende processor is 'Saxon' [Kaya]. Waarschijnlijk is deze processor zo populair omdat hij al van de eerste beginselen van XSLT meedraait. Toen versie 1.0 van XSLT nog in volle ontwikkeling was, volgde Saxon alle wijzigingen op de voet. Het resultaat hiervan was dat amper 17 dagen na de voorstelling van de XSLT 1.0 recommendation Saxon reeds heel deze standaard ondersteunde. Met gemiddeld 250 downloads op één dag, zal het niet gemakkelijk zijn om Saxon in te halen. Tot versie 8.0 van Saxon was het programma gratis. Sindsdien dient men te betalen voor updates en wordt er verder gegaan onder de naam Saxonica [Kayb].

Moderne webbrowsers beschikken tegenwoordig ook over een ingebouwde XSLT-processor (bvb Microsoft Internet Explorer 5.0). Wanneer je dan naar een XML-bestand surft, wordt de opmaak gedaan via een bijbehorende stylesheet.

1.4 XSLT 2.0

Na de publicatie van XSLT 1.0, traden er een aantal problemen op de voorgrond. Hiervan waren er een aantal die zo dringend een oplossing nodig hadden, dat de meeste XSLT-processoren zelf één bedachten. Hierdoor kreeg je echter processor-afhankelijk instructies, wat de standaardisatie niet ten goede kwam. Daarom besliste het W3C, alvorens een 2.0-versie van XSLT te publiceren, een XSLT 1.1 standaard uit te schrijven. Deze lostte de meest dringende problemen op, in afwachting van XSLT 2.0. Het grootste probleem dat erin behandeld werd, was het feit dat in de 1.0-versie result-tree-fragments niet meer als bomen werden beschouwd. Als je een dergelijke boom creëerde, kon deze op geen enkele manier de werking beïnvloeden. Een ander probleem is dat een XSLT 1.0-transformatie maar één output-document kan genereren. Dit is vervelend omdat een site gegenereerd uit een XML-bestand dan ook maar uit één pagina kon bestaan. Dit werd verholpen in versie 1.1.

Doordat de grootste problemen met XSLT 1.0 verholpen waren door een tussentijdse versie van XSLT uit te brengen, kon er verder gegaan worden met de ontwikkeling van XSLT 2.0. Buiten de aanpassingen van 1.0 in versie 1.1, is het de bedoeling om XSLT 2.0 enorm uit te breiden. De doelstellingen

werden reeds vastgelegd op 14 februari 2001. Één van deze doelstellingen is het nauwer samenwerken met XML-Schema [ST00], een taal om de structuur van een XML-document vast te leggen. Wanneer een transformatie gebeurt op een XML-bestand dat voldoet aan een bepaald XML-schema, is het de bedoeling dat XSLT informatie uit dat schema gaat gebruiken. Wanneer er bijvoorbeeld een waarde gelezen wordt, kan aan de hand van het schema het type achterhaald worden (bijvoorbeeld een datum). XSLT 2.0 zal dus veel meer types ondersteunen. Ook zullen er in versie 2.0 meer operaties op getallen en strings ingebouwd zijn. Een uitbreiding voor strings is het inlezen van bestanden die niet in XML geschreven zijn. In de nieuwe versie zijn er echter ook een aantal zaken die men absoluut niet wilt introduceren, hoewel er misschien bepaalde mensen voorstander van zijn. Ze gaan XSLT 2.0 niet ombouwen tot een general purpose programmeertaal (zoals Java of C++). Ook is het geen doelstelling om niet gestructureerde informatie om te zetten in XML-formaat. Op het moment van schrijven is XSLT 2.0 nog steeds een working draft, met als laatste update op 4 april 2005, wat toch aangeeft dat de nieuwe taal nog in volle ontwikkeling is.

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

 <xsl:template match="*">
     <S>
         <xsl:apply-templates select="." mode="makeBroadRepresentation"/>
     </S>
 </xsl:template>

 <xsl:template match="*" mode="makeBroadRepresentation">
     <!--zet de inputboom om in een brede voorstelling-->
     <xsl:choose>
         <!--we zitten NIET in een blad!-->
         <xsl:when test="./*">
             <xsl:call-template name="makeNode"/>
             <open/>
             <xsl:apply-templates select="./*" mode="makeBroadRepresentation"/>
             <close/>
         </xsl:when>

         <!--we zitten in een blad!-->
         <xsl:otherwise>
             <xsl:call-template name="makeNode"/>
         </xsl:otherwise>
     </xsl:choose>
 </xsl:template>

 <xsl:template name="makeNode">
     <!--maak een element van de huidige contentnode-->
     <xsl:choose>
         <xsl:when test="self::a">
             <a/>
         </xsl:when>
         <xsl:when test="self::b">
             <b/>
         </xsl:when>
         <xsl:when test="self::c">
             <c/>
         </xsl:when>
     </xsl:choose>
 </xsl:template>

</xsl:stylesheet>
```

Figuur 1.5: Stylesheet: van diepe naar brede boom

```
   <?xml version="1.0" encoding="utf-8"?>

<S>
    <a/>
    <open/>
    <b/>
    <open/>
    <c/>
    <close/>
    <c/>
    <open/>
    <a/>
    <b/>
    <close/>
    <close/>
</S>
```

Figuur 1.6: Brede boom als resultaat

```xml
<?xml version="1.0" encoding="UTF-8"?>

    <a>
        <b>
            <a/>
            <c/>
            <e/>
        </b>
        <b>
            <b/>
            <b>
                <a/>
                <a/>
                <a/>
            </b>
        </b>
        <b>
            <a/>
            <b>
                <c/>
                <b/>
                <d/>
            </b>
            <e/>
        </b>
    </a>
```

```xml
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:variable name="aantal">3</xsl:variable>
    <xsl:variable name="pos">2</xsl:variable>


    <xsl:template match="/">

        <xsl:for-each select="//b">
            <xsl:if test="count(./*) = $aantal">
                <xsl:copy-of select="./*[position() = $pos]"/>
            </xsl:if>
        </xsl:for-each>

    </xsl:template>

</xsl:stylesheet>
```

Figuur 1.7: Stylesheet en XML-bestand bij het derde voorbeeld

Hoofdstuk 2

Formele operationele semantiek

Een automaat is een formele specificatie van een algoritme. In theoretische informatica 1 zagen we een heel aantal automaten, zoals de deterministische eindige automaat, de niet-deterministische eindige automaat, de push-downautomaat enzovoort. Wanneer we deze formalismes op een afstand bekijken, zien we eigenlijk steeds dezelfde onderdelen terugkomen. Er worden eerst een aantal configuraties gedefinieerd. Een transitiefunctie bepaalt hoe we van de ene configuratie in de andere terechtkomen. Verder is er altijd een startconfiguratie van waaruit iedere berekening start. Een uitvoering van een automaat op een bepaalde inputstring is dus pad van configuraties. Wanneer een dergelijke uitvoering stopt in een accepterende configuratie, zeggen we dat de inputstring aanvaard wordt door de automaat. De verzameling van alle strings die aanvaard worden door de automaat, wordt de taal van de automaat genoemd.

In dit hoofdstuk zullen we beginnen met het herhalen van de meest krachtige automaat, de Turing machine. Daarna kijken we welke formalismen er bestaan die werken op bomen. De belangrijkste bronnen die hiervoor geraadpleegd werden waren [Sip96] en hoofdstukken 1 en 6 uit [CDG+97].

2.1 Woordtalen en woordtransformaties

2.1.1 De Turing machine

definitie 2.1. Een *Turing machine* is een zeven-tupel $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ waarbij Q, Σ en Γ eindige verzamelingen zijn en

- Q een verzameling toestanden is,
- Σ het input-alfabet is, dat het blancosymbool $\sqcup$ niet bevat,

- Γ het tape-alfabet is, met $\Sigma \subseteq \Gamma$ en $\sqcup \in \Gamma$,
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ de transitiefunctie is,
- $q_0 \in Q$ de starttoestand is,
- $q_{accept} \in Q$ de aanvaardende toestand is en
- $q_{reject} \in Q$ de verwerpende toestand is, waarbij $q_{accept} \neq q_{reject}$.

Bij de transitiefunctie δ kunnen we aan de hand van de huidige toestand en het huidige symbool bepalen naar welke toestand we overgaan. Het huidige symbool komt van de tape. Hierop bevindt zich namelijk een lees- en schrijfkop. Het huidige symbool is het symbool op de tape op de positie van deze lees- en schrijfkop. Wanneer we naar de volgende toestand overgaan, kunnen we dat symbool overschrijven. Verder specificeren we ook of we de kop links of rechts laten bewegen. Rechts bewegen kan, doordat de tape onbeperkt kan groeien, nooit een probleem veroorzaken. Bij het links bewegen is er een probleem wanneer de kop zich al uiterst links bevindt. Er wordt dan aangenomen dat de kop niet beweegt. We zeggen dat een Turing machine een inputstring aanvaardt van zodra hij in de aanvaardende toestand komt. Zodra de Turing machine in de verwerpende toestand komt, stopt de uitvoering zonder de string te aanvaarden. Tijdens de uitwerking worden dus telkens drie belangrijke paramaters bijgehouden: de positie van de lees- en schrijfkop, de huidige toestand en de volledige tape-inhoud. Een setting van deze drie parameters noemen we een configuratie.

definitie 2.2. Zij $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ een Turing machine. Een *configuratie van* M is een setting van de vorm uqv, waarbij $uv \in \Gamma^*$ de huidige tape-inhoud en $q \in Q$ de huidige toestand is. Het eerste symbool van v is het symbool onder de lees- en schrijfkop.

Het uitvoeren van één stap in een Turing machine komt overeen met het overgaan van de huidige configuratie naar een nieuwe, waarbij de transitiefunctie gerespecteerd wordt.

definitie 2.3. Zij $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ een Turing machine. De *step-relatie* $\to_M$ *van* M is gedefinieerd als volgt: zij $a, b, c \in \Gamma$, $u, v \in \Gamma^*$ en $q_i, q_j \in Q$. Zij $C_1 = uaq_ibv$ en $C_2 = uq_jacv$ twee configuraties van M. Er geldt dan $C_1 \to_M C_2$ als en slechts als er een transitie $\delta(q_i, b) = (q_j, c, L)$ bestaat. Zij $C_1' = uaq_ibv$ en $C_2' = uacq_jv$ twee configuraties van M. Er geldt: $C_1' \to_M C_2'$ als en slechts als er een transitie $\delta(q_i, b) = (q_j, c, R)$ bestaat.

Het eerste deel van deze definitie behandelt alle transities die links bewegen. Het tweede deel is voor de rechts-bewegende transities.

We zijn nu in staat naar een nieuwe configuratie over te gaan volgens de transitiefunctie van een Turing machine. Om te kunnen starten moeten we echter eerst nog de configuratie vastleggen waaruit we steeds vertrekken. De *startconfiguratie* van een Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ op input $w \in \Sigma^*$ is van de vorm $q_0 w$: we starten altijd in de starttoestand en de lees- en schrijfkop staat uiterst links. Een configuratie uqv is *aanvaardend* indien q de aanvaardende toestand q_{accept} is. Wanneer q gelijk is aan q_{reject} zeggen we dat de configuratie *verwerpend* is. Zowel de aanvaardende als de verwerpende toestand is een stoptoestand. Vanuit een stoptoestand kan geen nieuwe configuratie bereikt worden.

We hebben nu voldoende concepten ingevoerd om een volledige run van een Turing machine te definiëren.

definitie 2.4. Zij $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ een Turing machine. Een *run* van M op een inputstring $w \in \Sigma^*$ is een sequentie configuraties $C_1, C_2, \ldots, C_n$ zodat

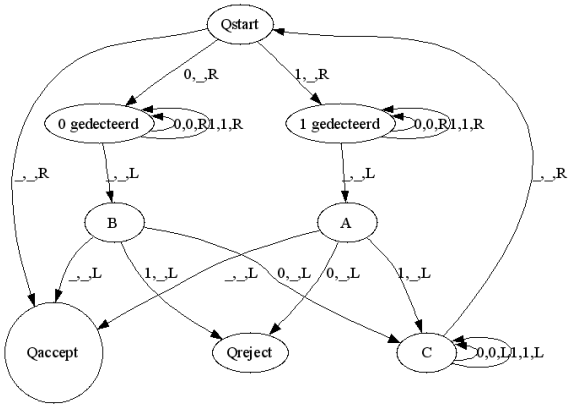
1. C_1 de beginconfiguratie is van M op w, dus $C_1 = q_0 w$,
2. voor iedere $i = 1, \ldots, n-1$ geldt $C_i \rightarrow_M C_{i+1}$,
3. C_n een stopconfiguratie is van M.

M aanvaardt de string w als en slecht als C_n de aanvaardende configuratie is.

Voorbeeld 2.1. We maken een Turing machine M die checkt of een string bestaande uit de symbolen '0' en '1' een palindroom is. Een palindroom is een woord dat je zowel van links naar rechts als van rechts naar links kunt lezen (bvb '010', '10101', '0', '', ...).

Een informele beschrijving van M is als volgt: M leest het eerste niet-lege symbool op de tape en vervangt dat door een blanco. In een toestand houdt hij bij of dit een '0' of een '1' is. Vervolgens wordt er gelopen tot het laatste niet-lege symbool van de tape. Indien dit een ander symbool is dan hetgeen er bijgehouden werd in de state, is de inputstring zeker geen palindroom. Indien het symbool hetzelfde is, vervangen we dit door een blancosymbool. De kop wordt terug naar het begin van de tape gebracht. Dit doen we net zolang totdat alle symbolen zijn gecontroleerd.

Figuur 2.1 geeft Turing machine M formeel weer ($\_$ stelt het blancosymbool voor). ■



Figuur 2.1: Turing machine die palindromen over '0' en '1' aanvaardt

We geven nu nog een stelling die we later in het hoofdstuk zullen gebruiken. De stelling vertelt ons dat we het aantal tapes van een Turing machine mogen uitbreiden.

stelling 2.1. Iedere multitape Turing machine heeft een equivalente Turing machine met één tape.

Bewijs. Zij M een k-multitape Turing machine, dit wil zeggen een Turing machine met k verschillende tapes. We creëren nu een single-tape Turing machine S die M simuleert. Het inputalfabet van S is gelijk aan dit van M. Het outputalfabet bevat dezelfde symbolen als dat van M, maar zal verder nog uitgebreid worden. De simulatie gebeurt in drie stappen. Op input $w = w_1 \dots w_n$:

1. S verdeelt zijn tape in k stukken, die elk een representatie zijn van één tape van M. Hij bakent iedere tape af met het symbool # (we gaan er vanuit dat # niet tot het tape-alfabet van M behoort). Om de koppositie van een tape van M bij te houden, breiden we het tape-alfabet van S uit. Ieder symbool komt nu ook voor met een $\bullet$ boven. Op deze manier duiden we aan op welke positie een lees- en schrijfkop zou staan in M. De tape van S ziet er dan initieel als volgt uit:

$$\#\dot{w}_1 w_2 \dots w_n \# \dot{\sqcup} \# \dot{\sqcup} \# \dots \#$$

2. Om één stap van M te simuleren scant S eerst zijn tape. Hij onthoudt alle symbolen op de aangeduide koppositie (aangegeven met een $\bullet$). Vervolgens wordt de tape een tweede keer afgelopen en worden tegelijk

de aanpassingen van koppositities en tapeinhouden aangebracht, zoals aangegeven in de transitiefunctie van M.

3. Wanneer S ergens een beweging naar rechts doet en dan op een # belandt, weet hij dat hij die tape moet uitbreiden. Hij schrijft dan een blancosymbool en verschuift alle cellen vanaf dit # één positie naar rechts. Wanneer S naar links beweegt en hij komt op een #, dan gaat hij terug achter het # staan.

□

2.1.2 Turing machine met output

Een Turing machine zoals we die tot nu toe gezien hebben, kan op een input enkel antwoorden met 'accept' of 'reject'. We kunnen echter de Turing machine uitbreiden zodat hij een string kan outputten. Hiervoor maken we een extra tape. In iedere stap van de Turing machine mogen we hier één symbool op schrijven en vervolgens zetten we de lees- en schrijfkop van deze bijkomende tape één positie naar rechts. Lezen kunnen we niet van deze tape. Wanneer de Turing machine in een aanvaardende toestand komt, is de output van de Turing machine de string die op de extra tape staat. Omdat de Turing machine, net zoals altijd, werkt op een inputstring, hebben we nu eigenlijk een stringtransformator. Dergelijke transformators zijn echter niet krachtiger dan een gewone Turing machine. We werken namelijk met twee tapes en uit stelling 2.1 weten we dat deze uitbreiding een Turing machine niet krachtiger maakt.

definitie 2.5. Een *stringtransformator* is een zeven-tupel $(Q, \Sigma, \Gamma, \delta,\ q_0,\ q_{accept},\ q_{reject})$ waarbij Q, Σ en Γ eindige verzamelingen zijn en

- Q een verzameling toestanden is,
- Σ het input-alfabet is, dat het blancosymbool $\sqcup$ niet bevat,
- Γ het tape-alfabet is, met $\Sigma \subseteq \Gamma$ en $\sqcup \in \Gamma$
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\} \times \Gamma$ de transitiefunctie is,
- $q_0 \in Q$ de starttoestand is,
- $q_{accept} \in Q$ de aanvaardende toestand is en
- $q_{reject} \in Q$ de verwerpende toestand is, waarbij $q_{accept} \neq q_{reject}$.

Het enige verschil in deze definitie met de definitie van een Turing machine, is dat de transitiefunctie uitgebreid is. Er moet nu nog een extra symbool gespecificeerd worden dat op de output-tape geschreven wordt. We zullen nu de werking formeel beschrijven.

definitie 2.6. Zij $S = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ een stringtransformator. Een *configuratie van* M is een setting van de vorm $uqv \cdot o$, waarbij $uv \in \Gamma^*$ de inhoud is van de eerste tape, $q \in Q$ de huidige toestand is en $o \in \Gamma^*$ de inhoud is van de tweede tape. Het eerste symbool van v is het symbool onder de lees- en schrijfkop van de eerste tape.

definitie 2.7. Zij $S = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ een stringtransformator en zij $u, v, o \in \Gamma^*$, $a, b, y \in \Gamma$ en $q_i, q_j \in Q$. De *step-relatie* van S, $\rightarrow_S$, is gedefinieerd als volgt: zij $C_1 = uaq_ibv \cdot o$ en $C_2 = uq_jacv \cdot oy$ twee configuraties van M. Er geldt $C_1 \rightarrow_S C_2$ als en slechts als er een transitie $\delta(q_i, b) = (q_j, c, L, y)$ bestaat. Zij $C_1' = uaq_ibv \cdot o$ en $C_2' = uacq_jv \cdot oy$ ook twee configuraties van M. Er geldt $C_1' \rightarrow_S C_2'$ als en slechts als $\delta(q_i, b) = (q_j, c, R, y)$ een transitie is gedefinieerd door S.

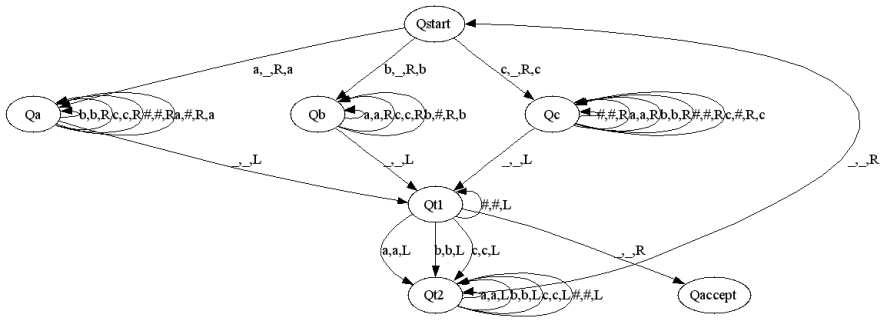
De startconfiguratie van een stringtransformator $S = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ op een input $w \in \Sigma^*$ is $q_0w \cdot \varepsilon$ (ε is de lege string). De aanvaardende configuratie is die configuratie waarin de toestand gelijk is aan q_{accept}. Wanneer de configuratie de toestand q_{reject} bevat, noemen we die configuratie verwerpend. Deze laatste twee types van configuraties zijn de stopconfiguraties.

definitie 2.8. Een *run* van een stringtransformator $S = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ op een input $w \in \Sigma^*$ is een sequentie configuraties $C_1, C_2, \ldots, C_n$ zodat

- $C_1 = q_0w \cdot \varepsilon$: de beginconfiguratie van S op w,
- voor iedere $i = 1, \ldots, n-1$ geldt $C_i \rightarrow_S C_{i+1}$,
- C_n een stopconfiguratie is van M.

Indien $C_n = uq_{accept}v \cdot o$ met $u, v, o \in \Gamma^*$, dan definiëren we de output van S op w als o. Wanneer C_n de verwerpende toestand bevat, is er geen output.

Voorbeeld 2.2. Als voorbeeld van een stringtransformatie geven we de Turing machine met output van figuur 2.2. Deze machine accepteert strings over $\{a, b, c\}$ en gaat vervolgens de symbolen groeperen volgens voorkomen in de input. Zo wordt *bbccabbaca* omgevormd tot *bbbbcccaaa* en *acccaccbca* tot *aaaccccccb*. De starttoestand is Q_{start}, de aanvaardende toestand is Q_{accept}. Hoe werkt nu deze Turing machine met output? Het eerste symbool wordt

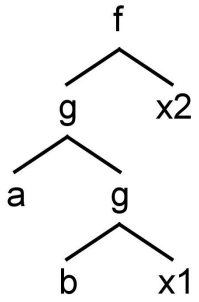


Figuur 2.2: Turing machine met output

gelezen en vervangen door een blanco (in de figuur voorgesteld met een _). Vervolgens wordt de input doorlezen en ieder symbool gelijk aan het eerste wordt naar de output geschreven en vervangen door een #. Andere symbolen blijven gewoon staan. Wanneer we op het einde van de input zijn, gaan we terug naar het begin met toestand Q_{t1}. Indien deze enkel #-symbolen leest tot aan een blanco, zijn alle symbolen van de input verwerkt en kunnen we stoppen. Het resultaat staat dan op de outputtape. Komen we echter nog een symbool a, b of c tegen, dan beginnen we terug van vooraf aan. Merk op dat er in de figuur ook transities zijn die niets schrijven op de outputtape. In theorie schrijven ze dan ε op de tape, maar omdat dat de figuur nog meer zou overladen, is dit symbool weggelaten. Ook hebben we de verwerpende toestand weggelaten omdat toch iedere string over $\{a, b, c\}$ output oplevert. ■

2.2 Boomtalen en boomtransformaties

We willen nu het accepteren en het omvormen van strings ook kunnen toepassen op bomen. We willen dus enerzijds een formalisme dat in staat is te bepalen of een boom tot een verzameling bomen (een boomtaal genoemd) behoort en anderszijds om, gegeven een boom, deze om te vormen tot een nieuwe boom. In dit hoofdstuk zullen we, in tegenstelling tot andere hoofdstukken, werken met ranked trees. Dit zijn bomen waar men aan de hand van de tag kan bepalen hoeveel kinderen er zijn. Het aantal kinderen van een knoop noemt men dan de ariteit. De knopen van een boom zijn dus gelabeld met een symbool uit een ranked alfabet. Dit is een tupel van de vorm $(\Sigma, \mathit{Ariteit})$, waarbij Σ een eindig alfabet van labels is en *Ariteit* een functie die aan iedere symbool uit Σ zijn ariteit toekent. De deelverzameling



Figuur 2.3: Boom uit $T(\Sigma, \chi)$

van symbolen uit Σ die allen ariteit p hebben, wordt genoteerd met Σ_p. Σ_0 wordt de verzameling constanten genoemd.

Zij χ een verzameling constanten, de variabelen genoemd, zodat $\Sigma_0 \cap \chi = \emptyset$. We definiëren de verzameling bomen $T(\Sigma, \chi)$ als:

$$\Sigma_0 \subseteq T(\Sigma, \chi),$$

$$\chi \subseteq T(\Sigma, \chi),$$

$$\forall p \geq 1, f \in \Sigma_p, t_1, \ldots, t_p \in T(\Sigma, \chi) : f(t_1, \ldots, t_p) \in T(\Sigma, \chi)$$

$T(\Sigma)$ stelt de verzameling bomen voor zonder variabelen (m.a.w. $\chi = \emptyset$). Indien iedere variabele hooguit één keer voorkomt in een boom, noemen we die boom *lineair*.

Voorbeeld 2.3. Zij $\Sigma = \{f, g, a, b\}$ een alfabet. De ariteit van f en g is twee, terwijl a en b constanten zijn (hun ariteit is nul). Zij $\chi = \{x_1, x_2\}$ een verzameling variabelen. Een boom uit de verzameling $T(\Sigma, \chi)$ staat getekend in figuur 2.3. ■

Wat waarschijnlijk nog niet duidelijk is, is het nut van de variabelen. Het idee is echter heel vergelijkbaar met het concept uit de wiskunde. x speelt in bijvoorbeeld de formule $f(x) = x+4$ de rol van een variabele. We kunnen nu die x een bepaalde waarde laten aannemen. Dit idee wordt nu doorgetrokken tot bomen. Zij $\chi_n = \{x_1, \ldots, x_n\}$ een verzameling van n variabelen en zoals gewoonlijk is Σ een ranked alfabet. Zij $C \in T(\Sigma, \chi_n)$ een lineaire boom en $t_1, \ldots, t_n \in T(\Sigma)$ (merk op dat deze bomen t_i geen variabelen bevatten). Met $C[t_1, \ldots, t_n]$ noteren we het vervangen van variabele x_i door t_i $(1 \leq i \leq n)$. Deze bekomen boom zit zeker ook in $T(\Sigma)$. De verzameling bomen met variabelen gekozen uit de verzameling $\{x_1, x_2, \ldots, x_n\}$ duiden we aan met $\mathcal{C}^n$ $(\mathcal{C}^1 = \mathcal{C})$.

2.2.1 Boomautomaten

We zullen nu een boomautomaat definiëren. Deze zal volgens hetzelfde idee werken als een stringautomaat. We gaan van configuratie naar configuratie door de input te lezen en houden ons daarbij aan de transitieregels. Een groot verschil met eerder geziene automaten is dat er geen starttoestand bestaat. Deze zit verwerkt in de transitieregels, zoals het eerste voorbeeld waarschijnlijk duidelijk zal maken.

definitie 2.9. Een *eindige boomautomaat* over een ranked alfabet Σ is een tupel (Q, Σ, Q_f, δ) waarbij Q een eindige verzameling toestanden is, $Q_f \subseteq Q$ de verzameling aanvaardende toestanden is en δ een verzameling transities is van de vorm:

$$f(q_1(x_1), \ldots, q_n(x_n)) \to q(f(x_1, \ldots, x_n))$$

met $n \geq 0$, $f \in \Sigma_n, q, q_1, \ldots, q_n \in Q, x_1, \ldots x_n \in \chi$.

Een automaat start in de bladeren van de boom. Hij werkt dan naar boven toe. Indien een boom t herleidbaar is naar een boom $q(t)$, met q een accepterende toestand, dan wordt de boom t aanvaard door de automaat. We moeten dan wel eerst definiëren wat een run is.

definitie 2.10. Zij $\beta = (Q, \Sigma, Q_f, \delta)$ een boomautomaat. De *step-relatie* $\to_\beta$ van β is gedefinieerd als: zij $t, t' \in T(\Sigma \cup Q)$

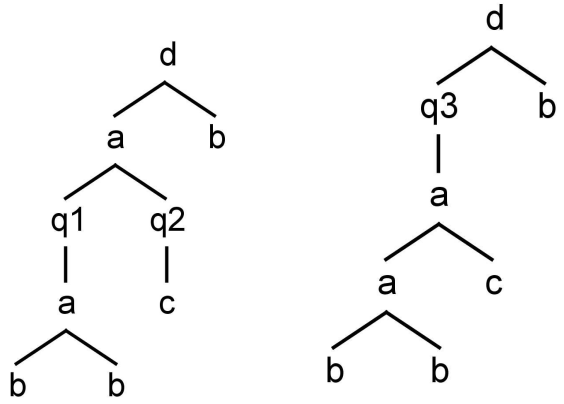
$$t \to_\beta t' \iff \begin{cases} \exists C \in \mathcal{C}(\Sigma \cup Q), \exists u_1, \ldots, u_n \in T(\Sigma) \\ \exists f(q_1(x_1), \ldots, q_n(x_n)) \to q(f(x_1, \ldots, x_n)) \in \delta \\ t = C[f(q_1(u_1), \ldots, q_n(u_n))] \\ t' = C[q(f(u_1, \ldots, u_n))] \end{cases}$$

We noteren de reflexief-transitieve sluiting van $\to_\beta$ als $\overset{*}{\to}_\beta$. Een *run* van β op een inputboom t is een sequentie bomen $t_1, t_2, \ldots, t_n$ uit $T(\Sigma \cup Q)$ zodat $t = t_1$ en voor iedere $i \in \{1, \ldots, n-1\}$ geldt $t_i \to_\beta t_{i+1}$. De boom t wordt aanvaard indien t_n een boom is van de vorm $q(f(u_1, \ldots, u_k))$, waarbij q een aanvaardende toestand is, m.a.w. $q \in Q_f$.

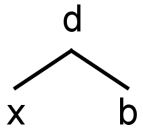
We beginnen eerst met een voorbeeldje om de werking van deze transitiefunctie te verklaren.

Voorbeeld 2.4. Beschouw een boomautomaat (Q, Σ, Q_f, δ) met $Q = \{q_1, q_2, q_3\}$ en $\Sigma = \{a, b, c, d\}$, met a en d ariteit twee en b en c beide constanten. Stel dat er een transitie van de vorm

$$a(q_1(x_1), q_2(x_2)) \to q_3(a(x_1, x_2))$$



Figuur 2.4: Bomen waarvan we de geldige overgang willen aantonen



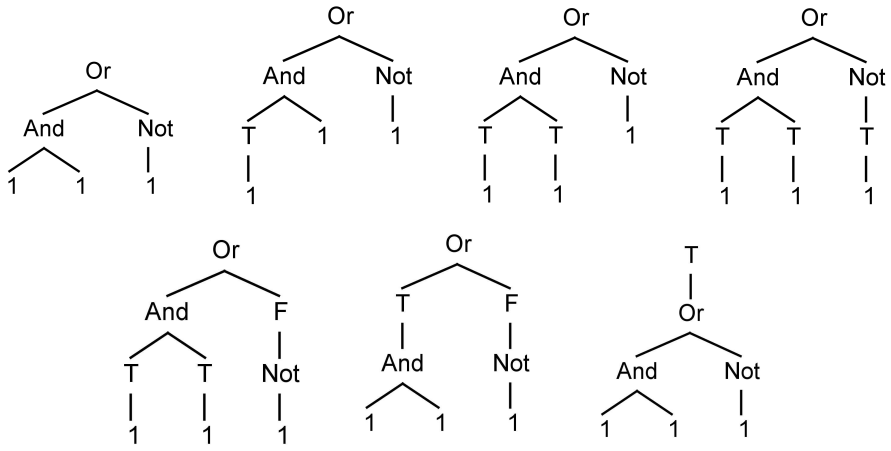
Figuur 2.5: Boom C

bestaat. Beschouw nu de twee bomen uit figuur 2.4.

We willen aantonen dat de overgang van de linkse boom naar de rechtse geldig is. We moeten dus een C vinden die voldoet aan de definitie van een run. C is een boom over $(\Sigma \cup Q)$ met één variabele in. Omdat we de bovenstaande transitieregel willen toepassen, moeten er ook twee bomen u_1 en u_2 bestaan (de ariteit van a is twee). Deze vullen we in op de plaats van de variabelen van de transitiefunctie, m.a.w. we vervangen x_1 door u_1 en x_2 door u_2. Omdat we de bovenstaande transitiefunctie willen gebruiken, volgt onmiddellijk dat u_1 gelijk is aan de boom $a(b, b)$ en u_2 gelijk is aan de boom c. We krijgen dan volgende instantie van de transitieregel:

$$a(q_1(a(b, b)), q_2(c)) \rightarrow q_3(a(a(b, b), c)).$$

We creëren nu de boom C als de boom in figuur 2.5. Hierbij speelt X de rol van een variabele. Wanneer we deze X vervangen door de boom aan de linkerkant van de transitie, krijgen we de linkse boom uit figuur 2.4. Indien we X vervangen door de boom aan de rechterkant van de transitie, krijgen we de rechtse boom uit figuur 2.4. We mogen dus besluiten dat de overgang geldig is. ■



Figuur 2.6: Binair circuit dat aanvaard wordt, samen met zijn afleiding

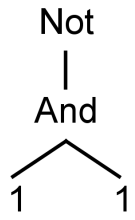
Voorbeeld 2.5. In dit voorbeeld wordt een volledige beschrijving van een boomautomaat A gegeven, samen met een run. Het idee is dat de automaat enkel bomen aanvaardt die een geldig binair circuit voorstellen. Een geldig binair circuit is er één dat evalueert tot 'true'. Zij $A = (Q, \Sigma, Q_f, \delta)$. Hierbij is $\Sigma = \{0, 1, And, Or, Not\}$. 0 en 1 zijn constanten, terwijl And en Or binaire symbolen zijn. Not is een unair symbool. We hebben twee toestanden T en F. Het idee is dat een knoop de toestand T krijgt toegewezen indien de deelboom geworteld in die knoop een geldig binair circuit voorstelt. De enige aanvaardende toestand is T. δ bestaat uit volgende regels:

$-0 \to F(0)$	$-Not(T(x_1)) \to F(Not(x_1))$
$-1 \to T(1)$	$-Not(F(x_1)) \to T(Not(x_1))$
$-And(T(x_1), T(x_2)) \to T(And(x_1, x_2))$	$-And(T(x_1), F(x_2)) \to F(And(x_1, x_2))$
$-And(F(x_1), T(x_2)) \to F(And(x_1, x_2))$	$-And(F(x_1), F(x_2)) \to F(And(x_1, x_2))$
$-Or(T(x_1), T(x_2)) \to T(Or(x_1, x_2))$	$-Or(T(x_1), F(x_2)) \to T(Or(x_1, x_2))$
$-Or(F(x_1), T(x_2)) \to T(Or(x_1, x_2))$	$-Or(F(x_1), F(x_2)) \to F(Or(x_1, x_2))$

Beschouw nu de bomen gegeven in figuur 2.6. Deze vormen geldige overgangen (gelezen van links naar rechts). De boom wordt dus aanvaard door A. Figuur 2.7 geeft een boom die niet geaccepteerd wordt. ■

2.2.2 Boomvertalers

We hebben nu een formalisme dat ons in staat stelt bepaalde boomtalen te aanvaarden. Het is echter ook mogelijk om een mechanisme uit te werken dat een gegeven boom vertaalt naar een nieuwe boom. We beginnen met een



Figuur 2.7: Binair circuit dat niet aanvaard wordt

formele definitie van een boomvertaler (tree transducer) en een run van een dergelijke automaat. Daarna gaan we door met weer eerst een voorbeeldje van een overgang te geven, waarna een volledige uitwerking volgt. Bij tree transducers heb je twee mogelijkheden. Er zijn er die starten in de bladeren van een boom en zo naar boven werken tot aan de wortel, deze worden bottum-up genoemd. Er kan ook gestart worden in de wortel en zo naar beneden gewerkt worden. Deze worden top-down-tree-transducers genoemd.

bottum-up boom vertalers

definitie 2.11. Een *bottum-up-tree-transducer* is een tupel $(Q, \Sigma, \Sigma', Q_f, \delta)$ waarbij Q een verzameling toestanden is, Σ en Σ' niet-lege verzamelingen symbolen zijn, de input- en outputsymbolen, $Q_f \subseteq Q$ de verzameling aanvaardende toestanden is en tenslotte δ een verzameling transities is, waarvan elk element één van volgende vormen heeft:

$$f(q_1(x_1), \ldots, q_n(x_n)) \to q(u)$$

waarbij $f \in \Sigma_n, u \in T(\Sigma', \chi_n), q, q_1, \ldots, q_n \in Q$ of

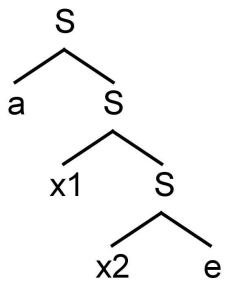
$$q(x_1) \to q'(u)$$

waarbij $u \in T(\Sigma', \chi_1), q, q' \in Q$.

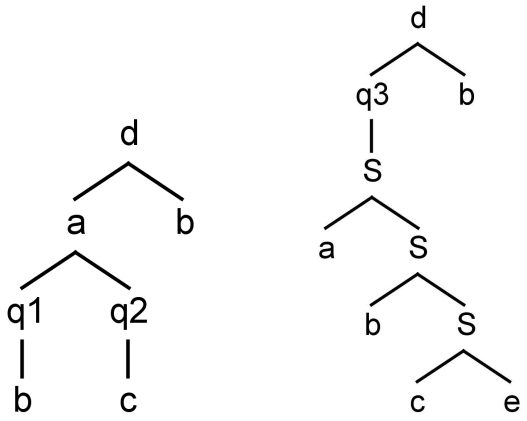
Net zoals een boomautomaat is er geen starttoestand. Deze zit in de transities verwerkt.

definitie 2.12. Zij $U = (Q, \Sigma, \Sigma', Q_f, \delta)$ een bottum-up-tree-transducer en zij $t, t' \in T(\Sigma \cup \Sigma' \cup Q)$. De *step-relatie* $\to_U$ van U is gedefinieerd als:

$$t \to_U t' \iff \begin{cases} \exists f(q_1(x_1), \ldots, q_n(x_n)) \to q(u) \in \delta \\ \exists C \in \mathcal{C}(\Sigma \cup \Sigma' \cup Q) \\ \exists u_1, \ldots, u_n \in T(\Sigma') \\ t = C[f(q_1(u_1), \ldots, q_n(u_n))] \\ t' = C[q(u\{x_1 \leftarrow u_1, \ldots, x_n \leftarrow u_n\})] \end{cases}$$



Figuur 2.8: Boom u



Figuur 2.9: Bomen waarvan we de geldige overgang willen aantonen

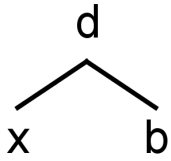
(Hierbij is $x_i \leftarrow u_i$ de notatie om variabele x_i te vervangen door u_i, $0 \leq i \leq n$.) We noteren de reflexief-transitieve sluiting van $\rightarrow_U$ als $\overset{*}{\rightarrow}_U$. Een *run* van een boomautomaat U op een boom t naar een boom t' is een sequentie stappen zodat $t \overset{*}{\rightarrow}_U q(t')$, waarbij q een aanvaardende toestand is.

Voorbeeld 2.6. Zij $U = (Q, \Sigma, \Sigma', Q_f, \delta)$ een bottum-up-tree-transducer, waarbij $\Sigma = \{a, b, c, d\}$ met b en c constanten en a en d binaire symbolen. $\Sigma' = \{a, b, c, d, e, S\}$, met d en S binaire symbolen en de overige symbolen zijn constanten. Q bevat drie toestanden, q_1, q_2, q_3. Voorts is

$$a(q_1(x_1), q_2(x_2)) \rightarrow q_3(u)$$

een transitie van δ, met u de boom uit figuur 2.8.

We willen nu aantonen dat de linkerboom van figuur 2.9 herleidbaar is naar de rechterboom. We moeten dus het bestaan aantonen van u_1, u_2 en



Figuur 2.10: Boom C

C. Figuur 2.10 geeft de boom C. u_1 is de constante b, u_2 is c. Wanneer we X in C vervangen door de het linkerdeel van de transitie, krijgen we de linkerboom van figuur 2.9. Vervangen we in de boom u de variabelen x_1 en x_2 door respectievelijk u_1 en u_2 en gebruiken we vervolgens dit resultaat om X in de boom C te vervangen, krijgen we de rechterboom van 2.9. Hiermee hebben we aangetoond dat we een geldige overgang hebben. ■

Boomtransformaties worden veel in de praktijk toegepast. Uiteraard is XSLT een boomtransformatie, maar ook bij het compilen van bijvoorbeeld programmeercode worden boomtransformaties gebruikt. Het verwerken van een Latex-bestand kan ook gebeuren met boomtransformaties. Het volgend voorbeeldje geeft een klein deeltje weer van waar een boomtransformatie gebruikt kan worden. De compiler krijgt als input een formule bestaande uit drie waardes (a, b en c). De enige toegestane operatoren zijn de (binaire) optelling en vermenigvuldiging. Voorbeelden hiervan zijn $+(a, b)$ en $\times(b, +(c, d))$. We kunnen deze prefixnotatie beschouwen als een boom. De vraag is nu hoeveel registers we nodig hebben om het eindresultaat van een formule te bekomen. Een register is een klein stukje geheugen waar een computerprocessor zeer snel toegang tot heeft en waarin hij alle waardes zet die hij nodig heeft bij een berekening.

Voorbeeld 2.7. De boomautomaat M die het minimaal aantal registers voor de uitwerking van een formule berekent is het tupel $(Q, \Sigma, \Sigma', Q_f, \delta)$, waarbij $Q = q_1, \ldots, q_n$ en $F = Q$ (we nemen aan dat de processor n registers heeft). Het alfabet bestaat uit de constanten a, b en c en de binaire operatoren $+$ en $\times$. Het output alfabet bevat dezelfde constanten, maar heeft als binaire symbolen $\overset{+}{}_i$ en $\overset{\times}{}_i$, $1 \leq i \leq n$. Hierbij stelt $\overset{+}{}_i$ een optelling voor die nood heeft aan i registers. We definiëren nu δ. Hierin worden nieuwe symbolen en toestanden tijdens de uitvoering berekend. Het zou kunnen dat we dan een symbool of toestand met onderschrift $n+1$ krijgen. Als dit voorkomt, nemen we aan dat de uitwerking stopt. De inputformule valt dan toch niet te berekenen met het beschikbaar aantal registers. Tijdens de latere

uitvoering van het programma (waar we dus het resultaat van de formule berekenen), berekenen we altijd eerst de linkeroperand van de optelling of vermenigvuldiging.

- $a \to q_0(a)\ b \to q_0(b)\ c \to q_0(c)$

  a, b of c wordt ingelezen en bijgehouden door de processor. Het getal moet dan ook niet opgeslagen worden in een register.

- $+(q_i(x), q_i(y)) \to q_{i+1}({}^{+}_{i+1}(x, y)) \qquad \times(q_i(x), q_i(y)) \to q_{i+1}({}^{\times}_{i+1}(x, y))$

  We moeten twee getallen optellen (vermenigvuldigen) die elk in i registers kunnen berekend worden. We kunnen dan het eerste getal in i registers berekenen en het resultaat ervan opslaan in het $i+1$-ste register. We hebben dan nog i registers vrij waarin we het tweede getal kunnen berekenen.

- $(i > j)\ +(q_i(x), q_j(y)) \to q_i({}^{+}_{i}(x, y)) \quad \times(q_i(x), q_j(y)) \to q_i({}^{\times}_{i}(x, y))$

  We reserveren voor de optelling (vermenigvuldiging) i registers, wat zeker voldoende is: het eerste getal kunnen we berekenen in i registers. Het resultaat slaan we op in één register. We hebben dan nog $i-1$ registers over voor de berekening van de tweede operand. Omdat deze in j registers kan gebeuren en omdat j strikt kleiner is dan i, zijn $i-1$ registers zeker voldoende.

- $(i < j)\ +(q_i(x), q_j(y)) \to q_j({}^{+}_{j}(y, x))\ \times(q_i(x), q_j(y)) \to q_j({}^{\times}_{j}(y, x))$

  Omdat zowel de optelling als de vermenigvuldiging commutatief zijn, kunnen we de twee deelbomen gewoon omdraaien.

Het probleem of een formule berekenbaar is met n registers, konden we ook oplossen met een boomautomaat. Indien deze zou eindigen in een aanvaardende toestand, is de formule berekenbaar. Bij de transformatie hebben we een extra voordeel dat we onze formule kunnen hervormen (zoals in het laatste puntje van δ), zodat deze later gemakkelijk omgegoten kan worden naar assemblercode (dit wordt in een volgend voorbeeld ook gedaan). ■

top-down boomvertalers

De bottum-up tree transducers werken, zoals de naam al liet vermoeden, vanuit de bladeren naar de wortel van een boom toe. De omgekeerde werking is echter ook mogelijk. Startend in de root begint een top-down tree transducer een boom te herschrijven totdat hij alle bladeren behandeld heeft. We geven eerst de formele definitie, waarna enkele voorbeelden volgen.

definitie 2.13. Een *top-down tree transducer* is een tupel $D = (Q, \Sigma, \Sigma', Q_s, \delta)$ waarbij Q een verzameling toestanden is, Σ en Σ' eindige, niet-lege verzamelingen input- en outputsymbolen zijn en $Q_s \subseteq Q$ een verzameling starttoestanden is. δ is een verzameling transities, waarvan elk element één van de volgende vormen heeft:

$$q(f(x_1, \ldots, x_n)) \to u[q_1(x_{i_1}), \ldots, q_p(x_{i_p})]$$

waarbij $f \in \Sigma_n, u \in \mathcal{C}^p(\Sigma'), q, q_1, \ldots, q_p \in Q, x_{i_1}, \ldots, x_{i_p} \in \chi_n$ of

$$q(x) \to u[q_1(x), \ldots, q_p(x)]$$

waarbij $u \in \mathcal{C}^p(\Sigma'), q, q_1, \ldots, q_p \in Q, x \in \chi$.

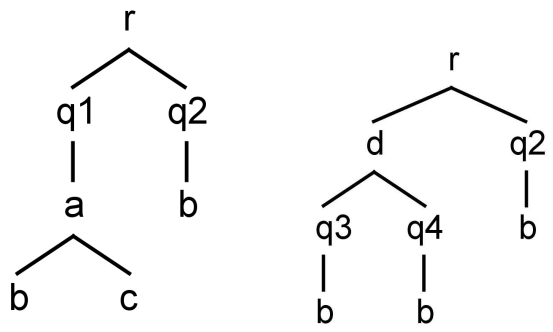
De bottum-up tree transducer had geen starttoestand: hij startte in de bladeren en wanneer er boven de wortel van de boom een toestand was toegevoegd werd er gestopt. Wanneer deze toestand in de verzameling aanvaardende toestanden zat, was de transformatie afgelopen. Hier draaien we alles om. Omdat we beginnen in de wortel, weten we nog niks over de boom. We moeten dus een starttoestand afspreken. Bij de bottum-up wisten we bij de start de inhoud van de bladeren al. Een top-down tree transducer stopt zodra hij een blad verwerkt heeft. Omdat er onder de bladeren niets meer komt, kunnen er ook geen veranderingen meer gebeuren en stopt uiteraard de transformatie. Vandaar dat er ook geen aanvaardende toestanden worden vermeld in de definitie. We gaan nu verder met te specificeren hoe de transitiefunctie werkt.

definitie 2.14. Zij $D = (Q, \Sigma, \Sigma', Q_s, \delta)$ een top-down tree transducer. De *steprelatie-relatie* $\to_D$ van D is gedefinieerd als:

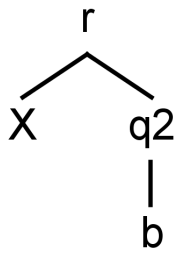
$$t \to_D t' \iff \begin{cases} \exists q(f(x_1, \ldots, x_n)) \to u[q_1(x_{i_1}), \ldots, q_p(x_{i_p})] \in \delta \\ \exists C \in \mathcal{C}(\Sigma \cup \Sigma' \cup Q) \\ \exists u_1, \ldots, u_n \in T(\Sigma) \\ t = C[q(f(u_1, \ldots, u_n))] \\ t' = C[u[q_1(v_1), \ldots, q_p(v_p)]] \qquad \text{waarbij } v_j = u_k \text{ als } x_{i_j} = x_k \end{cases}$$

We noteren de reflexief-transitieve sluiting van $\to_D$ als $\overset{*}{\to}_D$. Een *run* van D is een omzetting van een boom t in een boom t' zodat $q(t) \overset{*}{\to}_D t'$, waarbij q een starttoestand is.

Zoals gewoonlijk geven we een voorbeeldje om de transitie duidelijker te maken.



Figuur 2.11: Bomen waarvan we de geldige overgang willen aantonen



Figuur 2.12: Boom C

Voorbeeld 2.8. Zij $D = (Q, \Sigma, \Sigma', Q_s, \delta)$ een top-down tree transducer. Q bestaat uit de toestanden q_1, q_2, q_3, q_4. De starttoestand is q_1. Het inputalfabet bevat de constanten b en c en de binaire symbolen a en r. Het outputalfabet bevat dezelfde symbolen uitgebreid met een extra binair symbool d. Voorts bevat δ minstens de transitie

$$q_1(a(x_1, x_2)) \to d[q_3(x_1), q_4(x_1)]$$

We willen nu de overgang van de linkse boom in figuur 2.11 naar de rechtse aantonen. We moeten dus op zoek naar een C en een u_1 en u_2 zodat de overgang voldoet aan de definitie van een transitie. Als C vinden we de boom uit figuur 2.12. u_1 is de constante b en u_2 is c. Hieruit volgt dus dat de beoogde overgang geldig is. ■

In het voorgaande voorbeeld werd er een belangrijke eigenschap van top-down tree transducers toegepast. Het is namelijk mogelijk om de verschillende deelbomen van een knoop te kopiëren. Op deze manier kunnen we

die kopies elk op een andere manier behandelen door er een andere toestand aan toe te kennen. In het voorbeeld waren er twee kopies van deelboom b. Deze worden verder anders behandeld, omdat hun bovenliggende knoop een andere toestand bevat.

Voorbeeld 2.9. Dit voorbeeld werkt verder op voorbeeld 2.7. We hadden daar een formule in boomvorm gewijzigd in een boom die dezelfde formule bevatte, maar ook een attribuut in ieder knoop bijhield, namelijk het aantal registers om de bewerking uit te voeren. Nu geven we een volgende stap van de compiler. We zetten de gevonden formule om in echte assemblercode. Deze code bevat een (beperkt) aantal commando's die door een processor uitgevoerd kunnen worden. De processor beschikt (zoals iedere processor) over de mogelijkheid om één waarde bij te houden (in de accumulator). De overige waardes komen uit de registers. We nemen aan dat onze processor vier commandos heeft:

- Load i: laad de waarde van register i in de accumulator
- Store i: bewaar de huidige waarde van de accumulator in register i
- Add i: plaats in de accumulator de som van register i en de huidige waarde van de accumulator
- Mul i: plaats in de accumulator het product van register i en de huidige waarde van de accumulator

We geven nu de top-down tree transducer D die een output van de automaat in voorbeeld 2.7 omzet in codetaal. Zij dus $D = (Q, \Sigma, \Sigma', Q_s, \delta)$ waarbij het inputalfabet het outputalfabet van voorbeeld 2.7 is. Als outputalfabet hebben we de constanten (voor $1 \leq i \leq n$) S_i, A_i, M_i, L, a, b en c. Hierbij staat bvb. Si voor 'Store i'. Er is één binair symbool, namelijk C_2 en één symbool C_5 met rank 5 (van 'command'). De enigste toestand, wat dan ook meteen de starttoestand is, is q. Voorts beschikt D over de volgende transities:

$$
\begin{aligned}
&q(\overset{+}{i}(x, y)) \to C_5(q(x), S_i, q(y), A_i, S_i)\\
&q(\overset{\times}{i}(x, y)) \to C_5(q(x), S_i, q(y), M_i, S_i)\\
&q(a) \to C_2(L, a)\\
&q(b) \to C_2(L, b)\\
&q(c) \to C_2(L, c)
\end{aligned}
$$

■

2.3 XSLT en boomtransformaties

Wanneer we XSLT bekijken, valt het meteen op dat dit sterk gelijkt op een top-down tree transducer. Startend in de wortel van het XML-bestand gaat de XSLT-processor op zoek naar juiste templates die toepasbaar zijn. Uiteraard is de definitie van de top-down tree transducer zoals we hem hebben gezien in dit hoofdstuk te beperkt om volledig XSLT mee na te bootsen. In XSLT moet je onder andere al niet absoluut verdergaan met de kinderen te processen. In het werkelijkheid is XSLT een *tree-walking tree transducer with registers* [BMN02]. Om toch een gevoel te krijgen, geven we een klein stukje XSLT-code dat we omzetten in onze definitie van een top-down tree transducer. Omdat XML niet ranked is, zullen we aannemen dat het XML-fragment voldoet aan een DTD, waardoor we toch voor ieder element kunnen afleiden hoeveel kinderen er zijn.

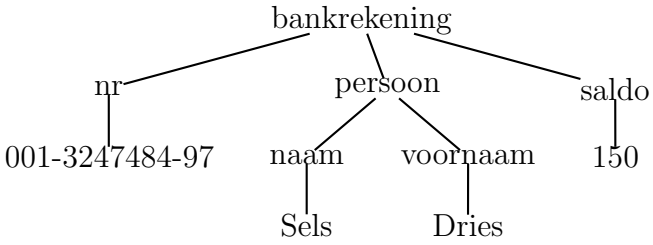
Voorbeeld 2.10. Beschouw het XML-bestand en het XSLT-fragment uit figuren 2.13 en 2.14. De stylesheet gaat informatie over een bankrekening publiceren in HTML-formaat (het resultaat is weergegeven in figuur 2.15). We proberen nu van dit XSLT-programma een top-down tree transducer te maken. We creëren als toestanden de verschillende modi van het programma, uitgebreid met een starttoestand q_{start}. Uit de boomvoorstelling kunnen we vlot afleiden wat ons inputalfabet is, samen met de rank van deze symbolen. De rekeningnummers, de namen en de verschillende saldo's van iedere bankrekenening is een oneindige verzameling van constanten. We kunnen echter deze tot een eindige beperken door gewoon alle nummers te nemen die een bankrekeningnummer voorstellen (dit zijn er een eindig aantal, want er bestaan maar een eindig aantal bankrekeningen), alle namen beperken tot alle namen die voorkomen in de wereld. Voor het saldo is er wel een probleem, we nemen echter aan dat we niet onder 0 kunnen gaan en een rekening slechts een maximaal bedrag kan bevatten. Dit zijn natuurlijk zeer gefrabiceerde beperkingen, maar het is dan ook maar een voorbeeld.

Tenslotte beschikken we over volgende transities

$$
\begin{aligned}
&q_{start}(\text{bankrekening(n, p, s)}) \to \text{html}(q_{ti}(\text{nr}), q_{nr}(\text{nr}), q_p(\text{p}), q_s(\text{s}))\\
&q_{ti}(\text{nr}(x)) \to \text{title}(x)\\
&q_{nr}(\text{nr}(x)) \to \text{h1}(x)\\
&q_p(\text{persoon}(n, vn)) \to \text{p}(q_n(n), \text{br}, q_{vn}(vn))\\
&q_n(\text{naam}(x)) \to \text{b}(x)\\
&q_{vn}(\text{voornaam}(x)) \to \text{i}(x)\\
&q_s(\text{saldo}(x)) \to \text{u}(x)
\end{aligned}
$$

■

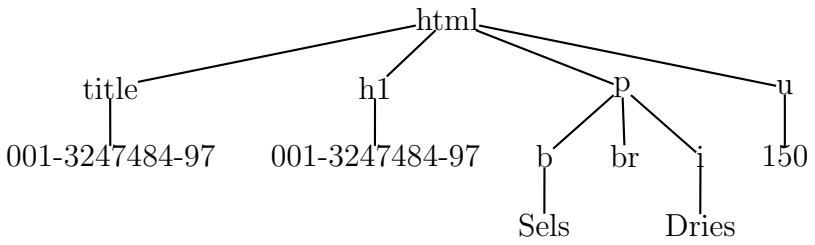
```
<?xml version="1.0" encoding="utf-8"?>

  <!DOCTYPE bankrekening [
      <!ELEMENT bankrekening (nr, persoon, saldo)>
      <!ELEMENT persoon (naam, voornaam)>
      <!ELEMENT nr (#PCDATA)>
      <!ELEMENT naam (#PCDATA)>
      <!ELEMENT voornaam (#PCDATA)>
      <!ELEMENT saldo (#PCDATA)>
  ]>


  <bankrekening>
      <nr>001-3247484-97</nr>
      <persoon>
          <naam>Sels</naam>
          <voornaam>Dries</voornaam>
      </persoon>
      <saldo>150</saldo>
  </bankrekening>
```



Figuur 2.13: Input XML-bestand

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

 <xsl:template match="/bankrekening">
     <html>
         <xsl:apply-templates select="nr" mode="qti"/>
         <xsl:apply-templates select="nr" mode="qnr"/>
         <xsl:apply-templates select="persoon" mode="qp"/>
         <xsl:apply-templates select="saldo" mode="qs"/>
     </html>
 </xsl:template>

 <xsl:template match="nr" mode="qti">
     <title><xsl:copy-of select="./node()"/></title>
 </xsl:template>

 <xsl:template match="nr" mode="qnr">
     <h1>
         <xsl:copy-of select="./node()"/>
     </h1>
 </xsl:template>

 <xsl:template match="persoon" mode="qp">
     <p>
         <xsl:apply-templates select="naam" mode="qn"/>
         <br/>
         <xsl:apply-templates select="voornaam" mode="qvn"/>
     </p>
 </xsl:template>

 <xsl:template match="naam" mode="qn">
     <b>
         <xsl:copy-of select="./node()"/>
     </b>
 </xsl:template>

 <xsl:template match="voornaam" mode="qvn">
     <i>
         <xsl:copy-of select="./node()"/>
     </i>
 </xsl:template>

 <xsl:template match="saldo" mode="qs">
     <u>
         <xsl:copy-of select="./node()"/>
     </u>
 </xsl:template>
</xsl:stylesheet>
```

Figuur 2.14: XSLT-bestand



Figuur 2.15: HTML-resultaatboom

Hoofdstuk 3

General replacement systems en subtree replacement systems

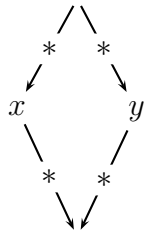
In het vorige hoofdstuk zagen we een aantal manieren om bomen te herschrijven, de zogenoemde tree transducers. In dit hoofdstuk worden subtree replacement systems besproken. Gegeven een boom en een aantal herschrijfregels, gaan we een deelboom van die boom vervangen door een nieuwe deelboom. Het hoofdstuk start met general replacement systems, wat een algemenere kijk geeft op herschrijfsystemen dan louter bomen. In de loop van het hoofdstuk worden een aantal belangrijke eigenschappen van herschrijfsystemen besproken en ook enkele stellingen bewezen. In het volgend hoofdstuk, waar we een formeel model voor XSLT uitwerken, zullen een heel deel resultaten uit dit hoofdstuk gebruikt worden om eigenschappen van XSLT mee aan te tonen.

Voor dit hoofdstuk werden twee papers gevolgd. Het deel over general replacement systems is gebaseerd op [Hue80]. Vanaf de twee paragraaf, waar we specifiek over bomen en boomherschrijvingen praten, is gebruik gemaakt van [Ros73].

3.1 General replacement systems

definitie 3.1. Zij B een verzameling en $\rightarrow$ een binaire relatie op B. Dan is het tupel $\beta = (B, \rightarrow)$ een *general replacement system* (GRS).

We bespreken enkele definities en eigenschappen van GRS's. Hiertoe nemen we aan dat we beschikken over een GRS $\beta = (B, \rightarrow)$. Indien we een tupel $(x, y) \in \rightarrow$ hebben, noteren we dit kortweg als $x \rightarrow y$. In de context van replacement systems, kunnen we dit intuïtief lezen als "herschrijf x in y".



Figuur 3.1: Confluentie

Voorbeeld 3.1. Zij B de verzameling van natuurlijke getallen. Zij $\rightarrow$ een relatie op B gedefinieerd als $\{(2^n, 2^{n+1}), (2^n, 3^n), (3^n, 3^{n+2})\}, \forall n \in B$.

Een mogelijke herschrijving is dan: $2^1 \rightarrow 2^2 \rightarrow 3^2 \rightarrow 3^4 \rightarrow \ldots = 2 \rightarrow 4 \rightarrow 9 \rightarrow 81 \rightarrow \ldots$. ■

We noteren de reflexief-transitieve sluiting van $\rightarrow$ met $\xrightarrow{*}$. Voorts introduceren we nog enkele afkortingen. Zij $x, y \in B$:

- $x \downarrow y \iff \exists z \in B : x \xrightarrow{*} z$ en $y \xrightarrow{*} z$
- $x \uparrow y \iff \exists z \in B : z \xrightarrow{*} x$ en $z \xrightarrow{*} y$

definitie 3.2. Een element t van B is *onvervangbaar* $\iff \forall s \in B : t \rightarrow s \Rightarrow t = s$.

t is een *normaalvorm* voor $r \in B \iff r \xrightarrow{*} t$ en t is onvervangbaar.

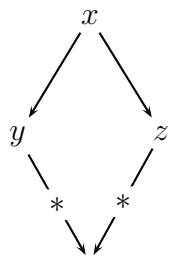
Een wenselijke eigenschap van een herschrijfsysteem is dat de normaalvorm van een element uniek is. Deze eigenschap is gekend onder de naam 'Church-Rosser'. De relatie $\rightarrow$ wordt dan ook wel confluent genoemd. We geven hier nu een formele definitie van:

definitie 3.3. De relatie $\rightarrow$ is *confluent* als en slechts als

$$\forall x, y \in B : x \uparrow y \Rightarrow x \downarrow y.$$

Een beeld hiervan is gegeven in figuur 3.1. We zien dat we verschillende transities volgen uit het topelement, maar we komen toch terug samen in éénzelfde element.

stelling 3.1. Als de relatie $\rightarrow$ confluent is, dan is de normaalvorm van ieder element, als het bestaat, uniek.



Figuur 3.2: Locale confluentie

Bewijs. Stel $x, t_1, t_2 \in B$ zodat t_1 en t_2 onvervangbaar zijn. Stel nu dat zowel t_1 als t_2 een normaalvorm is voor x. Er geldt dan $x \xrightarrow{*} t_1$ en $x \xrightarrow{*} t_2$. Doordat $\rightarrow$ confluent is, moet er dan ook een $z \in B$ bestaan zodat $t_1 \xrightarrow{*} z$ en $t_2 \xrightarrow{*} z$. Doordat t_1 en t_2 echter onvervangbaar zijn, moet in het bijzonder gelden $t_1 = z$ en $t_2 = z$ en dus $t_1 = t_2$. □

definitie 3.4. De relatie $\rightarrow$ is *locaal confluent* als en slechts als

$$\forall x, y, z \in B : x \rightarrow y \text{ en } x \rightarrow z \Rightarrow x \downarrow y.$$

De voorstelling hiervan is figuur 3.2. Het verschil met de definitie van confluentie is klein. Een voorbeeldje duidt het verschil aan.

Voorbeeld 3.2. Zij $(\mathbf{N}, \rightarrow)$ een GRS met $\rightarrow = \{(2^n, 2^{n+1}), (2^n, 3^n), (3^n, 3^{n+2})\}, \forall n \in \mathbf{N}$ (zie ook voorbeeld 3.1).

De relatie $\rightarrow$ is locaal confluent: $2^n \rightarrow 2^{n+1} \rightarrow 2^{n+2} \rightarrow 3^{n+2}$ en $2^n \rightarrow 3^n \rightarrow 3^{n+2}$. Het is echter niet confluent: $2 \rightarrow 3$ en $2 \rightarrow 4 \rightarrow 9$ en 3 en 9 zijn niet herleidbaar naar éénzelfde getal met de relatie $\rightarrow$. ■

definitie 3.5. We zeggen dat $\rightarrow$ *noetheriaans* is indien er geen oneindige sequentie $x_1 \rightarrow x_2 \rightarrow \ldots \rightarrow x_n \rightarrow \ldots$ $(n \in \mathbf{N})$ bestaat met $x_i \in B$, voor $i = 1, \ldots, n$.

lemma 3.1 (Newman's lemma)**.** Een noetheriaanse relatie is confluent als en slechts als ze locaal confluent is.

Bewijs. Zij $\rightarrow$ een noetheriaanse relatie. Het bewijs bestaat uit twee delen:

- $\rightarrow$ is confluent $\Rightarrow$ $\rightarrow$ is locaal confluent

  Zij $x, y, z \in B$ zodat $x \rightarrow y$ en $x \rightarrow z$. In het bijzonder geldt dan $x \xrightarrow{*} y$ en $x \xrightarrow{*} z$. Hierop kunnen we het gegeven dat $\rightarrow$ confluent is toepassen, namelijk er bestaat een $t \in B$ zodat $y \xrightarrow{*} t$ en $z \xrightarrow{*} t$. Hiermee is locale confluentie bewezen.

- $\rightarrow$ is locaal confluent $\Rightarrow$ $\rightarrow$ is confluent

  We beginnen het bewijs door met ieder element $x \in B$ zijn *hoogte* te associëren. Dit is de lengte van de langste herschrijfketting die je vanuit dat element kan maken. Doordat $\rightarrow$ noetheriaans is, weten we ook dat de hoogte van ieder element welgedefinieerd is.

  We tonen nu per inductie aan dat voor ieder natuurlijk getal n geldt dat voor ieder willekeurig element x van B met hoogte n geldt: als er een $y \in B$ en een $z \in B$ bestaat zodat $x \xrightarrow{*} y$ en $x \xrightarrow{*} z$, dan bestaat er ook een $t \in B$ zodat $y \xrightarrow{*} t$ en $z \xrightarrow{*} t$. Omdat in een noetheriaans systeem geldt dat iedere afleiding eindig is, hebben we confluentie bewezen voor B.

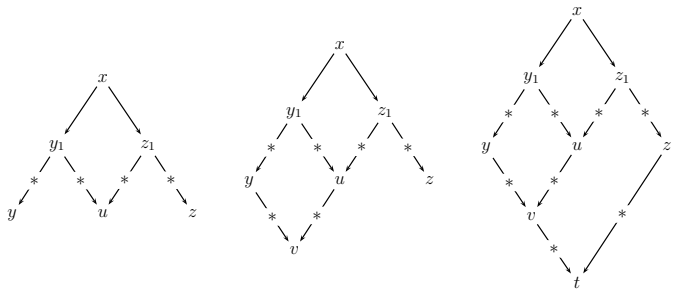
  Het basisgeval is wanneer n gelijk is aan 0. Zij $x, y, z \in B$ zodat $x \xrightarrow{*} y$ en $x \xrightarrow{*} z$ en de hoogte van x gelijk is aan 0. Er geldt dan in het bijzonder $x \xrightarrow{0} y$ en $x \xrightarrow{0} z$ en dus dat $x = y = z$. Neem nu $t \in B$ zodat $t = y = z$. Triviaal geldt dan $y \xrightarrow{*} t$ en $z \xrightarrow{*} t$, waarmee het basisgeval bewezen is.

  Stel nu dat het lemma is bewezen voor alle natuurlijke getallen strikt kleiner dan n. We tonen nu het lemma aan voor n. Zij $x \in B$ een willekeurig element met hoogte n en zij $y, z \in B$ zodat $x \xrightarrow{*} y$ en $x \xrightarrow{*} z$. We onderscheiden nu drie gevallen.

  Als y gelijk is aan x, dan stellen we t gelijk aan z. Uit $x \xrightarrow{*} z$ en $x = y$ volgt dan dat $y \xrightarrow{*} z$. We stelden z gelijk aan t, dus $y \xrightarrow{*} t$. Dat $z \xrightarrow{*} t$ volgt onmiddellijk uit de gelijkheid van z en t.

  Wanneer z gelijk is aan x, stellen we t gelijk aan y. We maken nu een analoge redenering als hiervoor. Uit de gelijkheid van t en y volgt meteen $y \xrightarrow{*} t$. Uit $x \xrightarrow{*} y$ volgt dan ook $z \xrightarrow{*} y$ en bijgevolg ook $z \xrightarrow{*} t$.

  Het laatste geval is wanneer zowel y als z verschillend is van x. Bijgevolg bestaat er een y_1 en een z_1 zodat $x \rightarrow y_1 \xrightarrow{*} y$ en $x \rightarrow z_1 \xrightarrow{*} z$. Doordat $x \rightarrow y_1$ en $x \rightarrow z_1$ en het gegeven dat $\rightarrow$ locaal confluent is, bestaat er een $u \in B$ zodat $y_1 \xrightarrow{*} u$ en $z_1 \xrightarrow{*} u$ (figuur 3.3, eerste tekening). Ook weten we dat de hoogte van y_1 en z_1 ten hoogste $n - 1$ is. Als dit niet zo was, zou er een pad van lengte n bestaan uit y_1. y_1 is bereikbaar vanuit x met 1 stap. Bijgevolg zou er ook een pad van lengte $n + 1$ bestaan uit x, wat tegenstrijdig is met het gegeven dat n de hoogte van x is. Dezelfde redenering gaat op voor z_1. We kunnen dus de inductiehypothese op y_1 toepassen om te besluiten dat er een $v \in B$ moet bestaan zodat $y \xrightarrow{*} v$ en $u \xrightarrow{*} v$ (figuur 3.3, tweede tekening). We hebben nu $z_1 \xrightarrow{*} v$ en $z_1 \xrightarrow{*} z$, samen met het feit dat de hoogte van z_1



Figuur 3.3: Bewijs Newman's lemma, lemma 3.1

hooguit $n-1$ is. We passen nu een tweede maal de inductiehypothese toe: er moet een $t \in B$ bestaan zodat $v \xrightarrow{*} t$ en $z \xrightarrow{*} t$ (figuur 3.3, derde tekening). In het bijzonder geldt dan ook $y \xrightarrow{*} t$, waarmee confluentie bewezen is.

□

lemma 3.2. Zij $\beta = (B, \rightarrow)$ een GRS. Indien er een binaire relatie $\rightarrow_\perp$ op B bestaat zodat

$(\xrightarrow{*}_\perp) = (\xrightarrow{*})$ en

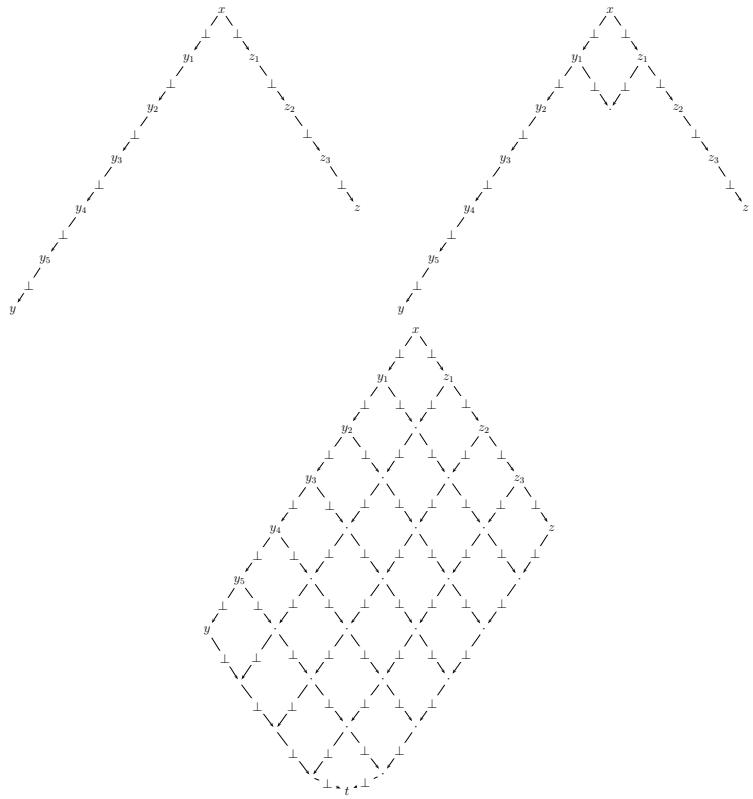
$(\forall x, y, z \in B)[(x \rightarrow_\perp y)$ en $(x \rightarrow_\perp z) \Rightarrow (\exists u \in B)(y \rightarrow_\perp u)$ en $(z \rightarrow_\perp u)]$

dan is β confluent.

Bewijs. Zij $x, y, z \in B$ zodat $x \xrightarrow{*} y$ en $x \xrightarrow{*} z$. We moeten een $t \in B$ vinden zodat $y \xrightarrow{*} t$ en $z \xrightarrow{*} t$. Omdat $(\xrightarrow{*}) \subseteq (\xrightarrow{*}_\perp)$ weten we $x \xrightarrow{*}_\perp y$ en $x \xrightarrow{*}_\perp z$. Zij m en n zodat $x \rightarrow_\perp y_1 \rightarrow_\perp y_2 \rightarrow_\perp \ldots \rightarrow_\perp y_m = y$ en $x \rightarrow_\perp z_1 \rightarrow_\perp z_2 \rightarrow_\perp \ldots \rightarrow_\perp z_n = z$. Door nu herhaaldelijk het gegeven toe te passen, kunnen we een t bekomen zodat $y \xrightarrow{n}_\perp t$ en $z \xrightarrow{m}_\perp t$. We tonen dit niet formeel aan, maar illustreren de werkwijze in figuur 3.4 voor $m = 6$ en $n = 4$. We hebben dus $y \xrightarrow{*}_\perp t$ en $z \xrightarrow{*}_\perp t$. Doordat gegeven is dat $(\xrightarrow{*}_\perp) \subseteq (\xrightarrow{*})$, geldt ook $y \xrightarrow{*} t$ en $z \xrightarrow{*} t$, waarmee confluentie aangetoond is. □

3.2 Bomen en substitutie

Zij A een verzameling. De verzameling van alle strings over A, dit zijn eindige sequenties van elementen van A, noteren we met A^*. In het bijzonder is $\mathbf{N}^*$ de



Figuur 3.4: Bewijs lemma 3.2

verzameling van strings over de natuurlijke getallen. Een string in A^* wordt voorgesteld door een opsomming (in haakjes en gescheiden door komma's) te geven van de elementen van A die de string vormen. De lengte van een string s wordt gegeven door $|s|$. De verzameling van strings in A^* met lengte j wordt genoteerd als A^j. Voor een string w en iedere $k \in \mathbf{N}$, is de string $k : w$ de string gevormd door de eerste k elementen van w en dit in dezelfde volgorde als in w (indien $k \geq |w|$ is $k : w$ w zelf). De concatenatie van twee strings wordt genoteerd met een $\cdot$. Tenslotte zijn de posities van een string w genummerd van 0 tot $|w| - 1$, waarbij $|w| - 1$ vaak afgekort wordt door -1.

definitie 3.6. Zij $m, n \in \mathbf{N}^*$. We definiëren een aantal relaties op $\mathbf{N}^*$.

- De vader-relatie

  $\mathrm{Fa}(n) = (|n| - 1) : n$, waarbij $n \neq ()$

- De linkerbroer-relatie

  $\mathrm{LBr}(n) = \mathrm{Fa}(n) \cdot (n_{-1} - 1)$, waarbij $n \neq ()$ en $n_{-1} \neq 0$

- De voorouder-relatie

  $m \,\mathrm{anc}\, n \iff |m| : n = m$

- de onafhankelijkheids-relatie (independence)

  $m \perp n \iff \neg(m \,\mathrm{anc}\, n \text{ of } n \,\mathrm{anc}\, m)$

De onafhankelijkheids-relatie kunnen we uitbreiden op verzamelingen.

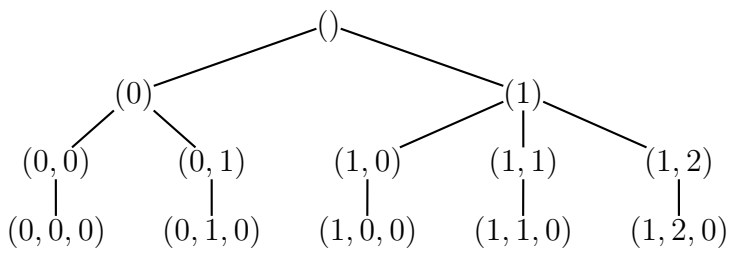
definitie 3.7. Zij $M, N \subseteq \mathbf{N}^*$. $M \perp N \iff (\forall m \in M)(\forall n \in N)(m \perp n)$

definitie 3.8. Een *boomdomein* is iedere eindige, niet-lege deelverzameling D van $\mathbf{N}^*$ die gesloten is onder de vader- en linkerbroer-relatie:

$$\mathrm{Fa}(D) \subseteq D \text{ en } \mathrm{LBr}(D) \subseteq D$$

Omdat dit allemaal vrij abstracte beschrijvingen zijn, volgt nu een voorbeeld.

Voorbeeld 3.3. Zij $D \subset \mathbf{N}^* = \{(), (0), (0,0), (0,0,0), (0,1), (0,1,0), (1), (1,0), (1,0,0), (1,1), (1,1,0), (1,2), (1,2,0)\}$. We willen nu aantonen dat D een boomdomein is. Eerst checken we de vader-relatie. Voor bv. $(1,1,0)$ is $\mathrm{Fa}((1,1,0))$ gelijk aan $(3-1) : (1,1,0) = 2 : (1,1,0) = (1,1)$. $(1,1)$ zit ook in D. We kunnen op gelijkaardige manier voor de andere elementen van D vaststellen dat de ouder van iedere knoop ook in D zit. Merk op dat de



Figuur 3.5: Boomvoorstelling voor boomdomein D

vader van () niet gedefinieerd is. We kijken nu nog de linkerbroer-relatie na. $\mathrm{LBr}((1,2)) = \mathrm{Fa}(1,2) \cdot (2-1) = (1) \cdot (1) = (1,1)$. $(1,1)$ is dus de linkerbroer van $(1,2)$ en deze zit ook in D. Merk op dat de linkerbroer van $(1,0)$ niet gedefinieerd is omdat $(1,0)_{-1} = 0$. We kunnen nu voor alle knopen nagaan dat hun linkerbroer in D zit. Dit klopt inderdaad, waardoor we mogen besluiten dat D een boomdomein is. Figuur 3.5 geeft een herkenbare weergave van de boom gedefinieerd door D. ■

We willen de verschillende knopen van een boom zoals gewoonlijk een label uit een verzameling labels V geven. We definiëren V_* als de verzameling van partiële functies die aan een element van een boomdomein een label uit V toekent.

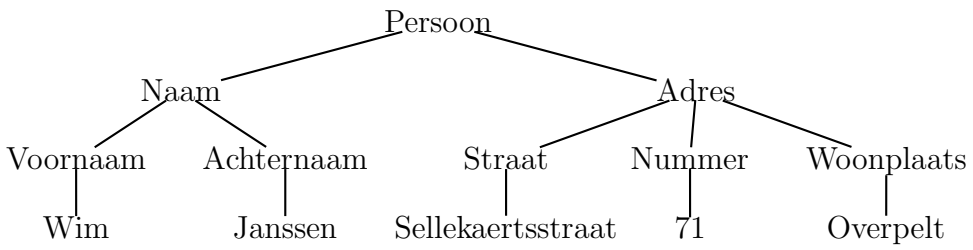
definitie 3.9. Zij V een willekeurige verzameling.

$V_* = \{R \mid R$ is een partiële functie $\mathbf{N}^* \to V$ en het domein van R is een boomdomein$\}$

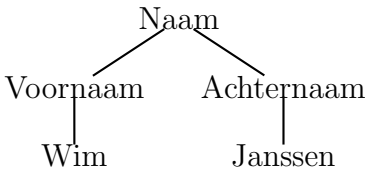
Voorbeeld 3.4. Zij $V = \{$Persoon, Naam, Voornaam, Achternaam, Adres, Straat, Nummer, Woonplaats, Wim, Janssen, Sellekaertsstraat, 71, Overpelt$\}$ en zij D het boomdomein uit voorbeeld 3.3. Een voorbeeldelement w van V_* is dan $w : D \to V$ zodat $w(()) =$ Persoon, $w((0)) =$ Naam, $w((1)) =$ Adres, ... Het resultaat hiervan is de boom weergegeven in figuur 3.6. Merk op dat de structuur van de boom niet gewijzigd is, enkel de knopen hebben een (nieuw) label. ■

We willen nu de herschrijving van een deelboom introduceren. Hiervoor moeten we een notatie overeenkomen om een deelboom van een boom te selecteren.

definitie 3.10. Zij V een verzameling en $R \in V_*$. Zij n een element uit het domein van R. We definiëren *de deelboom van R in knoop n*, genoteerd als R/n, als $\{(p,x) \mid (n \cdot p, x) \in R\}$.



Figuur 3.6: Gelabelde boom voor boomdomein D



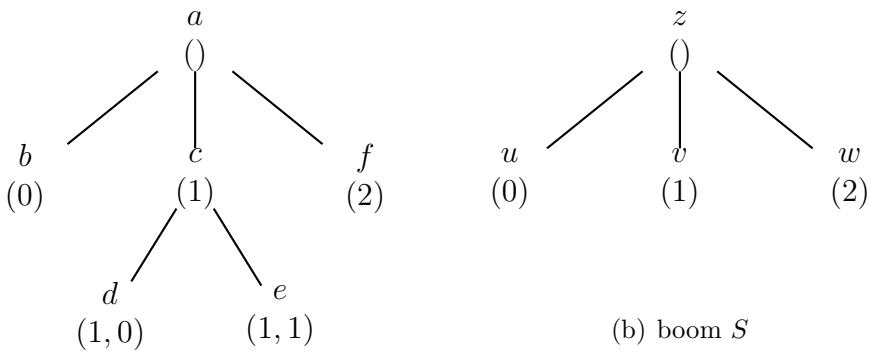
Figuur 3.7: Deelboom in knoop (0)

Voorbeeld 3.5. Beschouw opnieuw boomdomein D uit voorbeeld 3.4. De deelboom in knoop (0) is gelijk aan de koppels {((),Naam), ((0),Voornaam), ((1),Achternaam), ((0, 0),Wim), ((0, 1),Janssen)}. Wanneer we deze koppels als een boom tekenen, krijgen we figuur 3.7. ■

We gaan nu verder met de belangrijkste definitie in het hoofdstuk van subtree replacement systems, namelijk diegene waar we een deelboom vervangen door een nieuwe boom.

definitie 3.11. Zij V een verzameling en R en S elementen van V_*. Zij n een element van dom(R). We definiëren de *vervanging van knoop n door boom S in boom R*, genoteerd met $R(n \leftarrow S)$, als $\{(m, x)|(m, x) \in R$ en n is geen voorouder van $m)\} \cup \{(n \cdot p, y)|(p, y) \in S\}$

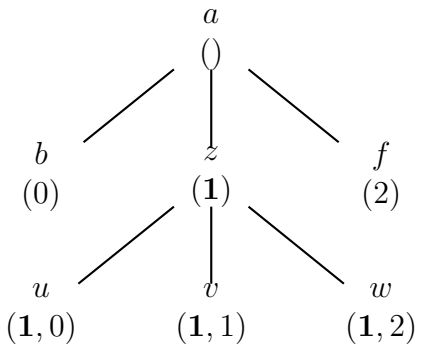
Voorbeeld 3.6. We geven een klein voorbeeldje van deze definitie. Beschouw de bomen uit figuur 3.8. We willen knoop (1) van boom R vervangen door boom S. Hiervoor verwijderen we eerst alle afstammelingen van deze knoop. Vervolgens voegen we aan dit resultaat alle knopen toe van S, maar eerst moet voor iedere nieuwe knoop zijn positie bepaald worden. Als voorbeeld nemen we knoop w uit S. Deze heeft positie (2) in S, maar in de resultaatboom krijgt hij positie $(1) \cdot (2) = (1, 2)$. Het resultaat van deze vervanging staat weergegeven in figuur 3.9. ■



(a) boom R

(b) boom S

Figuur 3.8: Het vervangen van een deelboom



Figuur 3.9: Het vervangen van een deelboom, resultaat

We gaan nu nog een aantal kleine lemma's opschrijven, om hier later gemakkelijk naar te kunnen verwijzen.

stelling 3.2. Zij V een verzameling en zij $R, S, T \in V_*$, $m, n \in \mathrm{dom}(R)$, $p \in \mathrm{dom}(S)$ en $m \perp n$.

- Embedding: $R(n \leftarrow S)/(n \cdot p) = S/p$
- Associativiteit: $R(n \leftarrow S(p \leftarrow T)) = R(n \leftarrow S)(n \cdot p \leftarrow T)$
- Persistentie: $R(n \leftarrow S)/m = R/m$
- Commutativiteit: $R(n \leftarrow S)(m \leftarrow T) = R(m \leftarrow T)(n \leftarrow S)$

Voor de lemma's waarbij een knoop een voorouder is van een andere knoop, definiëren we eerst nog het quotiënt tussen twee knopen.

definitie 3.12. Zij $m, n, p \in \mathbf{N}^*$. Het *quotiënt* van n door m, n/m, is p als en slechts als $m \cdot p = n$

Voorbeeld 3.7. Zij $n = (3, 4, 5, 6)$ en $m = (3, 4)$. Het quotiënt van n door m is gelijk aan $(5, 6)$. ■

stelling 3.3. Zij V een verzameling en $R, S \in V_*$. Zij $m, n \in \mathrm{dom}(R)$ zodat $m \,\mathrm{anc}\, n$. Er geldt:

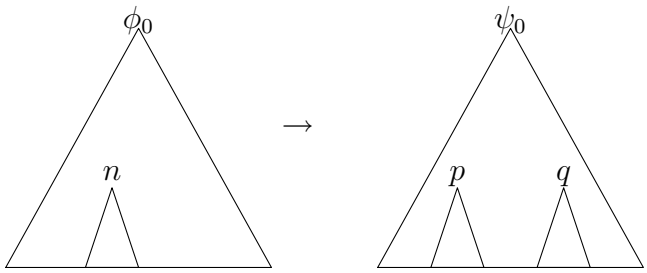
- cancellation: $R/n = (R/m)/(n/m)$
- distributiviteit: $R(n \leftarrow S)/m = (R/m)/(n/m \leftarrow S)$

3.3 Subtree replacement systems

Een speciaal geval van een general replacement system $\beta = (B, \rightarrow)$ is diegene waarbij we in iedere herschrijfstap een deelboom van een boom vervangen door een nieuwe boom. B is dan een verzameling bomen en $\rightarrow$ de relatie die een boom t_1 uit B herschrijft naar een nieuwe boom t_2 uit B. Zij n een knoop van t_1 en ϕ de deelboom geworteld in n. Wanneer we ϕ vervangen door een nieuwe boom ψ, noteren we dit als $\phi \rightarrow \psi$. We definiëren nu een subtree replacement system op een formele manier.

definitie 3.13. Een *subtree replacement system* (SRS) $\Re$ is een vier-tupel $(V, F, \rightarrow, \mathbf{R})$ waarbij

- V een verzameling is



Figuur 3.10: Residumap

- $F \subseteq V_*$
- $\mathbf{R} \subseteq V_* \times V_*$
- $(\forall T \in F)(\forall S \in V_*)(T \rightarrow S \iff (\exists \phi \rightarrow \psi \in \mathbf{R})(\exists n \in \mathrm{dom}(T))$ $[T/n = \phi \text{ en } S = T(n \leftarrow \psi)])$
- $(\rightarrow) \subseteq F \times F$

We zeggen dat $\Re$ *unequivocal* (of *deterministisch*) is wanneer $\mathbf{R}$ een partiële functie is.

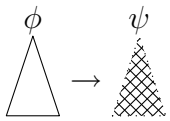
Wanneer we een boom herschrijven tot een nieuwe boom, vervangen we een deelboom ϕ_0 door een nieuwe boom ψ_0. Het kan zijn dat ϕ_0 zelf ook nog een deelboom ϕ bezat. De residumap van ϕ zijn de deelbomen van ψ_0 die gelijk zijn aan ϕ.

Voorbeeld 3.8. In figuur 3.10 hebben we de bomen ϕ_0 en ψ_0. Zij $\phi_0 \rightarrow \psi_0$ een herschrijfregel in een SRS. We zien dat na de herschrijving de deelboom ϕ geworteld in knoop n van ϕ_0 tweemaal herhaald wordt in ψ_0, namelijk geworteld in knoop p en in knoop q. We besluiten dat de residumap van n gelijk is aan $\{p, q\}$. ■

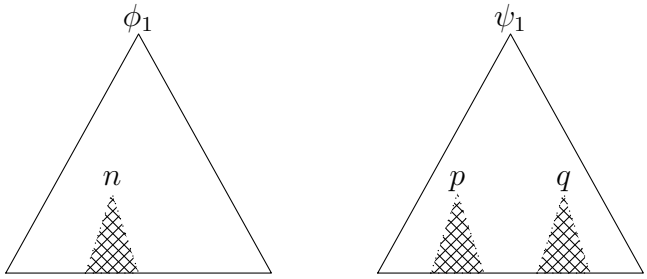
definitie 3.14. Zij $\Re = (V, F, \rightarrow, \mathbf{R})$ een SRS en zij $\phi_0 \rightarrow \psi_0 \in \mathbf{R}$. Een functie r gedefinieerd op $\mathrm{dom}(\phi_0)$ is een *residumap* voor $\phi_0 \rightarrow \psi_0$ als en slechts als

- $(\forall n \in \mathrm{dom}(\phi_0))[r(n) \subseteq \mathrm{dom}(\psi_0) \text{ en } (\forall m \in r(n))(\psi_0/m = \phi_0/n)]$
- $(\forall n_1, n_2 \in \mathrm{dom}(\phi_0))(n_1 \perp n_2 \Rightarrow r(n_1) \perp r(n_2))$

De residumap van $\phi_0 \rightarrow \psi_0$ in knoop n wordt genoteerd met $r[\phi_0, \psi_0](n)$



Figuur 3.11: $\phi \rightarrow \psi$



Figuur 3.12: ϕ_1 en ψ_1

We definiëren nu nog een belangrijke eigenschap, namelijk het gesloten zijn van een SRS. Alvorens een formele definitie te geven, maken we het concept duidelijk aan de hand van een voorbeeld.

Voorbeeld 3.9. Het voorbeeld gaat verder op voorbeeld 3.8. We stellen dat de regel $\phi \rightarrow \psi$ bestaat en dat deze visueel voorgesteld kan worden als figuur 3.11. We kunnen deze regel nu toepassen in ϕ_0 (namelijk op de positie van knoop n), het resultaat daarvan noemen we ϕ_1. Doordat de residumap van n de knopen p en q bevat, kunnen we dezelfde regel ook toepassen in ψ_0 (namelijk op de posities van knopen p en q). De uitwerking hiervan noemen we ψ_1. ϕ_1 en ψ_1 worden voorgesteld in figuur 3.12

Wanneer het SRS de herschrijfregel $\phi_1 \rightarrow \psi_1$ bevat, zeggen we dat het SRS gesloten is. ■

definitie 3.15. Zij $\Re = (V, F, \rightarrow, \mathbf{R})$ een SRS. We zeggen dat $\Re$ *gesloten* is als en slechts als het mogelijk is een residumap $r[\phi_0, \psi_0]$ toe te kennen aan iedere regel $\phi_0 \rightarrow \psi_0 \in \mathbf{R}$ op een zodanige manier dat: wanneer $\phi_0 \rightarrow \psi_0, \phi \rightarrow \psi \in \mathbf{R}$ en $n \in \mathrm{dom}(\phi_0)$, met $n \neq ()$ en $\phi_0/n = \phi$, dan is

(1) $\phi_1 \rightarrow \psi_1 \in \mathbf{R}$

(2) $(\forall p \in \mathrm{dom}(\phi_0))(p \perp n \Rightarrow r[\phi_1, \psi_1](p) = r[\phi_0, \psi_0](p))$

waarbij $\phi_1 = \phi_0(n \leftarrow \psi)$ en $\psi_1 = \psi_0(r[\phi_0, \psi_0](n) \leftarrow \psi)$[1]

[1]Merk op dat $r[\phi_0, \psi_0](n)$ meerdere knopen kan bevatten. Met deze notatie bedoelen we dan het vervangen van iedere knoop in die verzameling.

Puntje (2) van deze definitie zegt dat de residumap van een knoop p die onafhankelijk is van knoop n, hetzelfde blijft onder $\phi_1 \to \psi_1$. Dit zal nodig blijken in een later bewijs.

Stel we hebben een boom t. Zij N een verzameling onafhankelijke knopen van t, zodanig dat we een herschrijfregel hebben voor iedere knoop n van N. We kunnen dan t herschrijven door telkens een knoop van N te herschrijven. Omdat de herschrijvingen geen invloed op elkaar uitoefenen, zouden we echter ook alle knopen kunnen herschrijven in één stap. Hiervoor breiden we de herschrijfrelatie uit. Dit is wat de onderstaande definitie vertelt.

definitie 3.16. Zij $\Re = (V, F, \to, \mathbf{R})$ een SRS en $N \subseteq \mathbf{N}^*$. Zij $(n_0, \ldots, n_{|N|-1})$ de elementen van N. We definiëren de relatie $\to_N$ als de verzameling van alle elementen (R, S) in $F \times V_*$ zodanig dat:

N is een verzameling onafhankelijke knopen in $\mathrm{dom}(R)$

$(\exists \phi, \psi \in (V_*)^{|N|})[(\forall i < |N|)(R/n_i = \phi_i \text{ en } \phi_i \to \psi_i \in \mathbf{R}) \text{ en } S = R(n_0 \leftarrow \psi_0) \ldots (n_{-1} \leftarrow \psi_{-1})]$

We kunnen nu de relatie $(\to_\perp)$ definiëren als volgt: $(\to_\perp) = \bigcup_{N \subseteq \mathbf{N}^*} (\to_N)$

lemma 3.3. Zij $\to_\perp$ zoals in definitie 3.16. Er geldt: $(\to_N) \subseteq (\overset{|N|}{\to})$.

Bewijs. Zij N een verzameling onafhankelijke knopen zoals in definitie 3.16, zodat $R \to_N S$. Voor iedere knoop n_i in N is er een herschrijfregel $\phi_i \to \psi_i$. Voor iedere $i < |N|$ geldt:

$$R/n_i = R(n_0 \leftarrow \psi_0) \ldots (n_{i-1} \leftarrow \psi_{i-1})/n_i$$

Dit volgt rechtstreeks uit het feit dat N een verzameling van onafhankelijke knopen is en daardoor kunnen we i keer persistentie toepassen (stelling 3.2). We kunnen dus de regel $\phi_i \to \psi_i$ toepassen in knoop n_i. Dit komt overeen met:

$$R(n_0 \leftarrow \psi_0) \ldots (n_{i-1} \leftarrow \psi_{i-1}) \to R(n_0 \leftarrow \psi_0) \ldots (n_i \leftarrow \psi_i)$$

Hieruit volgt dan dat $(\to_N) \subseteq (\overset{|N|}{\to})$. □

stelling 3.4. Ieder gesloten, unequivocal SRS is confluent.

Bewijs. Zij $\Re = (V, F, \to, \mathbf{R})$ een gesloten, unequivocal subtree replacement system. We maken gebruik van lemma 3.2 en creëren de binaire relatie $\to_\perp$ zoals in definitie 3.16. Als we nu de twee eigenschappen uit lemma 3.2 kunnen aantonen, mogen we besluiten dat $\to$ confluent is.

Als eerste moet er aangetoond worden dat $\xrightarrow{*}_\perp = \xrightarrow{*}$. Uit het feit dat $\rightarrow_\perp$ gedefinieerd is als $\bigcup_{N \subseteq \mathbf{N}^*} (\rightarrow_N)$ en uit lemma 3.3 volgt $\rightarrow_\perp \subseteq \bigcup_{N \subseteq \mathbf{N}^*} (\xrightarrow{|N|})$. Uiteraard is dit laatste een deel van $\xrightarrow{*}$, dus mogen we besluiten dat $\rightarrow_\perp \subseteq \xrightarrow{*}$. In de andere richting geldt $\rightarrow \subseteq \bigcup_{N \subseteq \mathbf{N}^*, |N|=1} \rightarrow_N \subseteq \rightarrow_\perp \subseteq \xrightarrow{*}_\perp$. We kunnen dus concluderen dat $\xrightarrow{*}_\perp = \xrightarrow{*}$. Bijgevolg wordt er aan de eerste eis van lemma 3.2 voldaan.

De tweede eis van lemma 3.2 zullen we bewijzen door onderstaande formule aan te tonen.

$$(\forall R, S, S' \in F)(\forall N, N' \in \mathrm{dom}(R))$$

$$[(R \rightarrow_N S \wedge R \rightarrow_{N'} S' \wedge |N \cup N'| = K] \Rightarrow$$

$$(\exists P \subseteq \mathrm{dom}(S))(\exists P' \subseteq \mathrm{dom}(S'))(\exists T \in F)$$

$$(S \rightarrow_P T \wedge S' \rightarrow_{P'} T \wedge P \cup P' \subseteq (N \cup N') \cdot \mathbf{N}^*)]$$

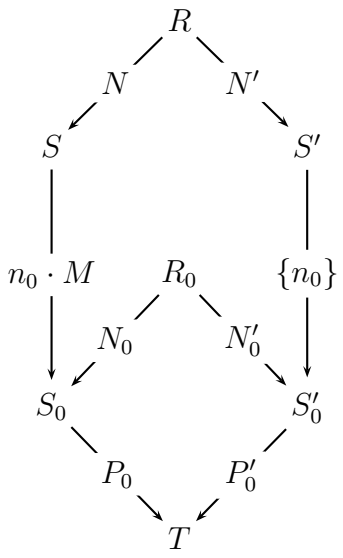
Het bewijs van deze formule gebeurt per inductie op K.

Het basisgeval is wanneer K gelijk is aan 0. We weten dan dat zowel de verzameling N als N' leeg is. Hieruit blijkt dat $R = S$ en $R = S'$. We kiezen een willekeurige $P \subseteq S$ zodat $S \rightarrow_P T$ geldt. We stellen P' gelijk aan P. Hieruit volgt $S' \rightarrow_{P'} T$, omdat S' gelijk is aan S. Omdat $N \cup N'$ leeg was, geldt er triviaal dat $P \cup P' \subseteq (N \cup N') \cdot \mathbf{N}^*$ (P is zeker een deelverzameling van $\mathbf{N}^*$).

Stel nu dat bovenstaande formule geldig is voor alle $k < K$. We bewijzen dan de stelling voor K. Zij R, S, S', N, N' zoals in de te bewijzen formule, zodanig dat $|N \cup N'| = K$. Zij $\phi_i \rightarrow \psi_i$ de regel toegepast in $n_i \in N$ om S te vormen. Analoog is de regel $\phi'_j \rightarrow \psi'_j$ toegepast in knoop $n_j \in N'$ om S' te vormen. We tonen nu aan hoe we P, P' en T kiezen. We hebben hier twee mogelijkheden.

Het kan zijn dat $N \perp N'$. We kiezen $P = N'$ en $P' = N$. We stellen T gelijk aan $R(n_0 \leftarrow \psi_0) \ldots (n_{-1} \leftarrow \psi_{-1})(n'_0 \leftarrow \psi'_0) \ldots (n'_{-1} \leftarrow \psi'_{-1})$. Door persistentie en commutativiteit uit stelling 3.2 te gebruiken, voldoen P, P' en T aan de te bewijzen formule.

Als echter anderszijds geldt dat $\neg(N \perp N')$, dan bestaat er een knoop uit N die een voorouder is van een knoop uit N'. (Het bewijs dat er een knoop uit N' een voorouder is van een knoop uit N is volledig analoog). We kunnen nu de knopen van N en N' zo hernummeren zodanig dat er een $J > 0$ bestaat zodat voor iedere $j < J$ geldt dat $n_0 \,\mathrm{anc}\, n'_j$ en voor iedere $j \geq J$ *niet* geldt dat $n_0 \,\mathrm{anc}\, n'_j$. We creëren nu de verzamelingen $N_0 = N - n_0$ en $N'_0 = N' - \{n'_j | j < J\}$. We kunnen nu aantonen dat $n_0 \perp (N_0 \cup N'_0)$: we



Figuur 3.13: Bewijs stelling 3.4

weten namelijk dat zowel N als N' verzamelingen zijn van onafhankelijke knopen. N_0 is dat dus zeker ook, dus er is al zeker geen knoop $n \in N_0$ zodat $n_0 \perp n$. Verder is N'_0 gedefinieerd als de verzameling van knopen in N' die onafhankelijk zijn van knoop n_0.

Het bewijs kan nu weer gesplitst worden in twee delen. Het kan namelijk zijn dat N' een knoop bevat die ook in N zit, maar het kan ook zijn dat $N \cap N'$ leeg is.

In het eerste geval bestaat er een n'_j met $j < J$ zodat $n'_j = n_0$. We stellen dan R_0 gelijk aan $R(n_0 \leftarrow \psi_0)$. Doordat $\Re$ unequivocal is, is R_0 ook gelijk aan $R(n'_j \leftarrow \psi'_j)$. We krijgen dus $R_0 \rightarrow_{N_0} S$ en $R_0 \rightarrow_{N'-\{n_j\}} S'$. Ook kunnen we besluiten dat $|N_0 \cup (N' - \{n'_j\})|$ gelijk is aan $K - 1$. We kunnen dan de inductiehypothese op K gebruiken. Er bestaat dus een P, P', T zodat $S \rightarrow_P T$ en $S \rightarrow_{P'} T$. Ook weten we uit de inductie dat $(P \cup P') \subseteq (N_0 \cup (N' - \{n'_j\})) \cdot \mathbf{N}^*$. Doordat deze laatste uitdrukking zeker een deelverzameling is van $(N \cup N') \cdot \mathbf{N}^*$, is de formule bewezen.

In het andere geval, waarbij dus voor iedere $n'_j, j < J$, geldt dat $n'_j \neq n_0$, is het idee om de inductiehypothese toe te passen op N_0 en N'_0. Hiervoor gaan we bomen R_0, S_0, S'_0 creëren zodat $R_0 \rightarrow_{N_0} S_0$ en $R_0 \rightarrow_{N'_0} S'_0$. Om dit te bereiken wordt een nieuwe regel $\bar{\phi} \rightarrow \bar{\psi}$ gemaakt. Het bewijs is gemakkelijker om volgen met figuur 3.13.

Zij $r[\phi_0, \psi_0]$ de residumap voor $\phi_0 \rightarrow \psi_0$ (dit is de regel toegepast in n_0). Omdat N' een verzameling onafhankelijke knopen is, is $\{n'_j/n_0 | j < J\}$ dat

ook. Uit het tweede deel van de definitie van residumap volgt dan dat iedere $M_j = r[\phi_0, \psi_0](n'j/n_0), j < J$, ook een onafhankelijke deelverzameling van $\mathrm{dom}(\psi_0)$ is. Voor $j, k < J, j \neq k$ geldt $M_j \perp M_k$. De unie van deze verzamelingen, $M = \bigcup_{j<J} M_j$, is dan ook een verzameling onafhankelijke knopen van $\mathrm{dom}(\psi_0)$.

We construeren nu twee bomen $\bar{\phi}$ en $\bar{\psi}$.

$$\bar{\phi} = \phi_0(n'_0/n_0 \leftarrow \psi'_0) \ldots (n'_{J-1}/n_0 \leftarrow \psi'_{J-1})$$

$$\bar{\psi} = \psi_0(M_0 \leftarrow \psi'_0) \ldots (M_{J-1} \leftarrow \psi'_{J-1})$$

Omdat $\Re$ gesloten is, volgt dat $\bar{\phi} \rightarrow \bar{\psi}$ ook in $\mathbf{R}$ zit. We creëren nu de bomen R_0, S_0, S'_0.

$$R_0 = R(n_0 \leftarrow \bar{\psi}), S_0 = S(n_0 \leftarrow \bar{\psi}), S'_0 = S'(n_0 \leftarrow \bar{\psi})$$

In het begin van het bewijs namen we aan dat de regel $\phi_0 \rightarrow \psi_0$ werd toegepast in n_0. Hieruit volgt dan onmiddellijk dat R/n_0 gelijk is aan ϕ_0 (anders zouden we die regel niet kunnen toepassen in n_0). We kunnen dan ook $\bar{\phi} = \phi_0(n'_0/n_0 \leftarrow \psi'_0) \ldots (n'_{J-1}/n_0 \leftarrow \psi'_{J-1})$ schrijven als $\bar{\phi} = (R/n_0)(n'_0/n_0 \leftarrow \psi'_0) \ldots (n'_{J-1}/n_0 \leftarrow \psi'_{J-1})$. Hierop kunnen we dan weer distributiviteit toepassen (stelling 3.3), waardoor we $\bar{\phi} = R(n'_0 \leftarrow \psi'_0) \ldots (n'_{J-1} \leftarrow \psi'_{J-1})/n_0$ krijgen. Doordat $n_0 \perp N'_0$ kunnen we $|N'_0|$ keer persistentie (stelling 3.2) toepassen op deze laatste formule voor $\bar{\phi}$. We mogen dus schrijven $\bar{\phi} = R(n'_0 \leftarrow \psi'_0) \ldots (n'_{J-1} \leftarrow \psi'_{J-1}) \ldots (n'_{-1} \leftarrow \psi'_{-1})/n_0$. Dit is hetzelfde als S'/n_0.

We hebben nu $\bar{\phi} = S'/n_0$, $\bar{\phi} \rightarrow \bar{\psi} \in \mathbf{R}$ en $S'_0 = S'(n_0 \leftarrow \bar{\psi})$. Hieruit volgt: $S' \rightarrow_{\{n_0\}} = S'_0$.

Omdat M een verzameling onafhankelijke knopen is van $\mathrm{dom}(\psi_0)$ en omdat $\psi_0 = S/n_0$ is $n_0 \cdot M$ een verzameling onafhankelijke knopen van $\mathrm{dom}(S)$. Zij nu $j < J$ en $p \in M_j$. We kunnen nu $|N| - 1$ keer persistentie en 1 keer embedding (stelling 3.2) toepassen, zodat

$$S/(n_0 \cdot p) = R(n_0 \leftarrow \psi_0) \ldots (n_{-1} \leftarrow \psi_{-1})/(n_0 \cdot p)$$

$$S/(n_0 \cdot p) = R(n_0 \leftarrow \psi_0)/(n_0 \cdot p)$$

$$S/(n_0 \cdot p) = \psi_0/p$$

Door de definitie van een residumap weten we dat ψ_0/p gelijk is aan $\phi_0/(n'_j/n_0)$. In deze knoop konden we de regel $\phi'_j \rightarrow \psi'_j$ toepassen, waaruit volgt dat $\psi_0/p = \phi'_j$ en dus $S/(n_0 \cdot p) = \phi'_j$. We weten dat S bekomen werd door n_0 in R te vervangen door ψ_0. Door nu nogmaals de knoop n_0 in S te herschrijven als ψ_0, doen we eigenlijk een overbodige herschrijving. We

mogen dus schrijven: $S(n_0 \leftarrow \psi_0) = S$. Door nu $|M|$ keer associativiteit toe te passen, krijgen we

$$S_0 = S(n_0 \leftarrow \bar{\psi})$$

$$S_0 = S(n_0 \leftarrow \psi_0(M_0 \leftarrow \psi'_0) \ldots (M_{J-1} \leftarrow \psi'_{J-1}))$$

$$S_0 = S(n_0 \leftarrow \psi_0)(n_0 \cdot M_0 \leftarrow \psi'_0) \ldots (n_0 \cdot M_{J-1} \leftarrow \psi'_{J-1})$$

$$S_0 = S(n_0 \cdot M_0 \leftarrow \psi'_0) \ldots (n_0 \cdot M_{J-1} \leftarrow \psi'_{J-1})$$

Hieruit kunnen we afleiden dat $S \rightarrow_{n_0 \cdot M} S_0$, waarbij $\phi'_j \rightarrow \psi'_j$ toegepast is in $n_0 \cdot p$, met $j < J$ en $p \in M_j$.

We bewijzen nu dat $R_0 \in F$. Door $\phi'_j \rightarrow \psi'_j$ in n'_j toe te passen voor iedere $j < J$ en vervolgens $\bar{\phi} \rightarrow \bar{\psi}$ toe te passen in n_0, krijgen we:

$$R \rightarrow_\perp R(n'_0 \leftarrow \psi'_0) \ldots (n'_{J-1} \leftarrow \psi'_{J-1}) \rightarrow$$

$$R(n_0 \leftarrow \bar{\psi}) = R_0$$

We kunnen aantonen dat $R_0 \rightarrow_{N_0} S_0$ door $\phi_i \rightarrow \psi_i$ toe te passen in n_i voor iedere $i, 0 < i < |N|$. Door in iedere knoop n'_j de regel $\phi'_j \rightarrow \psi'_j$, $J \leq j < |N'|$, tonen we aan dat $R_0 \rightarrow_{N'_0} S'_0$. We kunnen dan de inductiehypothese toepassen. We weten namelijk dat $|N_0 \cup N'_0| = K - 1 - J < K$. Hieruit krijgen we een P_0, P'_0, T met $S_0 \rightarrow_{P_0} T$, $S'_0 \rightarrow_{P'_0} T$ en $P_0 \cup P'_0 \subseteq (N_0 \cup N'_0) \cdot \mathbf{N}^*$.

Zij $P = n_0 \cdot M \cup P_0$ en $P' = \{n_0\} \cup P'_0$. Uit $S_0 \rightarrow_{P_0} T$ en uit $S \rightarrow_{n_0 \cdot M} S_0$ volgt $S \rightarrow_P T$. Uit $S'_0 \rightarrow_{P'_0} T$ en uit $S' \rightarrow_{n_0} S'_0$ volgt $S' \rightarrow_{P'} T$. $P \cup P' \subseteq n_0 \cdot \mathbf{N}^* \cup (N_0 \cup N'_0) \cdot \mathbf{N}^* \subseteq (N \cup N') \cdot \mathbf{N}^*$, waarmee het bewijs afgelopen is. □

Hoofdstuk 4

Abstracte syntax en formele semantiek voor XSLT 1.0

In dit hoofdstuk willen we XSLT 1.0 in een syntactisch en formeel model gieten. We zullen de W3C-specificatie [Cla99] definiëren als een subtree replacement system, een formalisme dat we uitvoerig hebben besproken in het vorige hoofdstuk. Hiermee kunnen we dan trachten bepaalde eigenschappen, zoals bijvoorbeeld confluentie, voor XSLT 1.0 aan te tonen.

We starten met een abstracte syntax te definiëren voor XSLT 1.0. Daarna kijken we, gegeven een syntactisch correct XSLT-programma, wat de betekenis is van dat programma. We bespreken met andere woorden de semantiek van XSLT. Belangrijk is dat we voor het onderzoek van XSLT een subset hebben genomen van de volledige XSLT 1.0 specificatie. Features zoals bijvoorbeeld string-operaties zijn weggelaten. Hoewel ze in de praktijk veel gebruikt worden, willen we ons hier vooral bezighouden met boomtransformaties.

4.1 Abstracte syntax van XSLT 1.0

XSLT 1.0 werkt nauw samen met XPath 1.0 [CD99]. We zullen ook hier een abstractie van maken. We nemen gewoon aan dat we over een patroontaal $\wp$ beschikken, zonder in detail te specificeren hoe die opgebouwd is. We geven wel de strikt noodzakelijke functies aan die gedefinieerd moeten zijn op $\wp$ en een aantal patronen waarover $\wp$ zeker beschikt. Een eerste functie op $\wp$ die we vermelden is $type : \wp \rightarrow \{\text{'teller', 'node-set'}\}$, die aangeeft of het resultaat van een patroon een teller dan wel een node-set is. De exacte betekenis van deze functie zal verder in de tekst duidelijk worden.

We beginnen nu met de abstracte syntax van XSLT 1.0 te beschrijven.

definitie 4.1. Een *XSLT-programma* Π is een tupel van de vorm $(V, type, R, <, <_g, def)$ waarbij

- V $= V_g \cup V_l$ een verzameling variabelen is, gepartitioneerd in twee delen: de globale variabelen[1] V_g en de locale variabelen V_l,
- $type : V \rightarrow \{teller, node - set, result - tree - fragment\}$,
- R een verzameling regels is,
- $<$ een totale orde is op R,
- $<_g$ een partïele orde is op V_g,
- *def* een functie is die aan elke globale variabele $v \in V_g$ een variabelestatement voor v toekent.

definitie 4.2. Een *regel* bestaat uit een template samen met:

- een naam, die uniek moet zijn binnen het programma

  of

- een pattern uit onze patroontaal $\wp$ van het type $node - set$ en een mode, dit is een string.

definitie 4.3. Een *template* is een eindige boom waarvan de root altijd het label 'template' draagt. De overige knopen zijn statementknopen of (gewone) knopen gelabeld met een symbool uit een eindig alfabet Σ.

De verschillende soorten statement-knopen zijn:

- een apply-templatesstatementknoop bestaande uit:
  - de label 'apply-templates', samen met een pattern, een mode en een eindige reeks $(v, expressie)$ paren, waarbij v een variabele is en *expressie* een pattern uit $\wp$ van het type $type(v)$ die de inhoud van v bepaalt.
  - De enige kinderen zijn with-paramstatementknopen.
- een variabelestatementknoop:

  wanneer het variabelestatement een variabele declareert van het type 'node-set' (respectievelijk 'teller'), bestaat de statementknoop uit de label 'node-set' (respectievelijk 'teller'), samen met de naam van de variabele en een expressie uit $\wp$ van het type $type(v)$. Er zijn dan geen

[1] In XSLT worden deze de top-level variabelen genoemd.

kinderen. Is de variabele echter van het type 'result-tree-fragment', dan bevat het label de string 'result-tree-fragment' samen met de variabelenaam. Het enige kind is een template.

- een with-paramstatementknoop:

  een with-paramstatementknoop declareert een parameter van het type 'result-tree-fragment'. Het label bestaat uit de speciale string 'with-param' en de naam van de parameter die gedeclareerd wordt. Het enige kind is een template.

- een call-templatestatementknoop bestaande uit:

  - de speciale label 'call-template', samen met een naam. Ook bevat het label een eindige reeks $(v, \mathit{expressie})$ paren, waarbij v een variabelenaam is en *expressie* een expressie uit $\wp$ is van het type $\mathit{type}(v)$ die de inhoud van v bepaalt.
  - De enige kinderen zijn with-paramstatementknopen.

- een if-thenstatementknoop dat bestaat uit:

  - de speciale label 'if-then', samen met een test. Deze test is een expressie van het type node-set uit $\wp$.
  - Het enige kind is een template.

- een choose-when-otherwisestatementknoop bestaande uit:

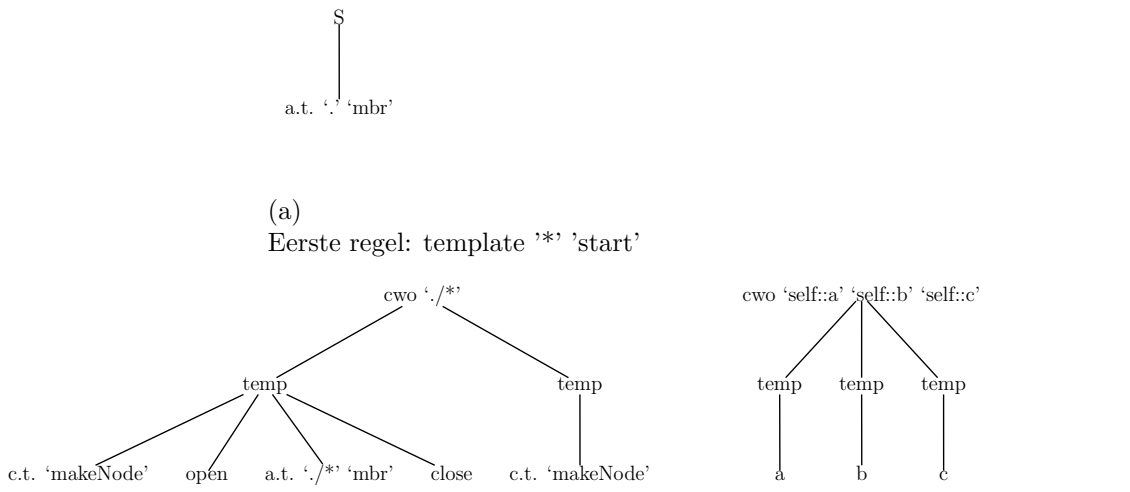
  - de speciale label 'choose-when-otherwise', samen met een genummerde lijst van testen. Iedere test is een expressie van het type node-set uit $\wp$.
  - Indien het aantal testen gelijk is aan n, dan zijn er n of $n + 1$ kinderen, die allemaal templates zijn.

- een for-eachstatementknoop dat bestaat uit:

  - de speciale label 'for-each', samen met een expressie uit $\wp$ van het type node-set.
  - Een template als kind.

- een copy-ofstatementknoop bestaande uit:

  - de speciale label 'copy-of', samen met een expressie uit $\wp$ van het type node-set of de naam van een variabele van het type result-tree-fragment.



(a)
Eerste regel: template '*' 'start'

(b) Tweede regel: template '*' 'mbr'

(c)
Derde regel: template 'makeNode'

Figuur 4.1: Van diepe naar brede boom in abstracte syntax

De if-then-, choose-when-otherwise- en for-eachstatementknopen zullen we *controlestatementknopen* noemen en de knopen die geen statementknoop zijn *constante knopen*. De niet-controlestatementknopen zijn de overige knopen, met name de apply-templates-, call-template-, variabele-, copy-of- en with-paramstatementknopen.

4.1.1 Uitgewerkt voorbeeld

Het XSLT-voorbeeld uit sectie 1.3.1 transformeerde een gegeven boom over het alfabet $\{a, b, c\}$ naar een boomvoorstelling waar haakjes gebruikt werden om de nesting voor te stellen. Figuur 4.1 geeft hetzelfde voorbeeld in onze nieuwe syntax, waarbij we wel enkele afkortingen hebben gebruikt om de bomen klein te houden. Merk op dat in onze syntax iedere regel die een pattern heeft, ook een mode krijgt. Zo krijgt de eerste regel dus ook een mode.

In de abstracte syntax is dit XSLT-programma dus het tupel $(V, \mathit{type}, R, <, <_g, \mathrm{def})$, waarbij V leeg is en hierdoor $<_g$ en def ook leeg zijn. Voor de relatie $<$ nemen we de volgorde van de regels hoe deze vermeld zijn in de figuur.

4.2 Formele semantiek van XSLT 1.0

We willen nu verder gaan met het beschrijven van de semantiek van een XSLT-programma: gegeven een XSLT programma in onze abstracte syntax, wat is de betekenis hiervan? Om dit te verklaren, beschouwen we een programma $\Pi(V, type, R, <, <_g, \text{def})$ en een eindig alfabet Σ. We beginnen met een aantal noodzakelijke hulpconcepten.

definitie 4.4. Zij t een boom over Σ met s knopen[2]. Een *waarde over t van het type 'node-set'* is een willekeurige verzameling knopen van t. Een *waarde over t van het type 'teller'* is een natuurlijk getal tussen 0 en s. Tenslotte hebben we nog een *waarde over t van het type 'result-tree-fragment'*. Dit is een rij eindige bomen over Σ.

definitie 4.5. Zij t een boom met s knopen. Een *variabeletoekenning over t* is een functie die aan elke $v \in V$ een waarde over t van het type $type(v)$ toekent.

definitie 4.6. Zij t een boom met s knopen. Een *context over t* is een tupel van de vorm (n, i, m, α) waarbij

- n een knoop is van t;
- $i \in \{1, \ldots, s\}$;
- $m \in \{1, \ldots, s\}$, met $m \geq i$;
- α een variabeletoekenning is over t.

De functie *eval* kent aan elk drietal (e, t, c), met $e \in \wp$, t een boom en c een context over t, een waarde over t van het type $type(e)$ toe. Voorts beschikt $\wp$ over de expressie *root* met volgende betekenis: $eval(root, t, c) = \{r\}$, waarbij r de root is van t. Ook bestaat de expressie *children*, zodat $eval(children, t, c) = \{n_1, n_2, \ldots, n_k\}$, waarbij $n_1, n_2, \ldots, n_k$ de kinderen zijn van de contextnode van c in t, opgesomd in pre-order van t. Belangrijk om te vermelden is dat variabelen van het type result-tree-fragment nooit een invloed kunnen hebben op het resultaat van *eval*.

definitie 4.7. Een *template is terminaal* als de template geen statement-knopen meer bevat.

definitie 4.8. Een *configuratie γ over een boom t* is een koppel (*template*, *context*) waarbij

[2]Een *boom over* Σ is een boom waarin alle knopen gelabeld zijn met een symbool uit Σ.

- *template* een template is,
- *context* een partiële functie is die aan bepaalde knopen van *template* een context over t toekent.

We noteren *template* als template(γ) en *context* als context(γ). Als γ een configuratie is waarbij de template terminaal is, noemen we γ ook terminaal en we noteren dan *template* als result(γ).

definitie 4.9. Zij t een boom. Zij $\gamma_1 = (template_1, context_1)$ een configuratie over t en zij s een controlestatementknoop van $template_1$, zodat $context_1(s)$ gedefinieerd is. We definiëren een nieuwe configuratie $\gamma_2 = (template_2, context_2)$ zodanig dat yield$(\gamma_1, s) = \gamma_2$.

- s is een if - thenstatementknoop.

  Zij *test* de expressie horende bij s, *ouder* de ouder van s in $template_1$ en $c := context_1(s)$ de context horende bij s. We stellen eval$(test, c, t)$ gelijk aan de node-set ns. We hebben twee mogelijkheden. Wanneer ns de lege verzameling knopen is, creëren we $template_2$ uit $template_1$ door knoop s te verwijderen. De context van de overige knopen wijzigt niet. Als echter $ns \neq \emptyset$, zij dan *template* de template geworteld in het unieke kind van s. *template* is van de vorm template(rij), waarbij rij de concatenatie is van de top-level deelbomen van *template*. Construeer nu uit $template_1$ $template_2$ door knoop s te vervangen door een kopie van rij, waarbij dus elke top-level deelboom van deze kopie als ouder de knoop *ouder* krijgt. Vervolgens definiëren we nog de functie $context_2$. Voor iedere knoop k die reeds voorkwam in $template_1$, wijzigt de context niet. Voor iedere nieuwe toegevoegde knoop k van $template_2$ geldt dat $context_2(k)$ gelijk is aan $context_1(k')$, waarbij k de kopie is van k'.

- s is een choose-when-otherwisestatementknoop.

  Zij *ouder* de ouder van s in $template_1$ en $c := context_1(s)$ de context horende bij s. Zij *test* de eerste in de genummerde lijst van n expressies zodanig dat eval$(test, c, t)$ niet gelijk is aan de lege verzameling knopen. Stel dat *test* volgnummer i heeft, stel dan *template* gelijk aan het i-de kind van s. Indien *test* niet gevonden kan worden, stel dan *template* gelijk aan het $n+1$-ste kind van s. We behandelen nu eerst het geval dat *template* niet gedefinieerd is, m.a.w. geen enkele test voldeed en er was geen $n+1$-ste kind. We construeren dan $template_2$ uit $template_1$ door knoop s te verwijderen. De context voor de overige knopen wijzigt niet. Als echter *template* wel berekend kon worden, is deze altijd van de vorm template(rij). Construeer nu uit $template_1$ $template_2$ door

knoop s te vervangen door een kopie van *rij*, waarbij dus elke top-level deelboom van deze kopie als ouder de knoop *ouder* krijgt. Vervolgens definiëren we nog de functie $context_2$. Voor iedere knoop k die reeds voorkwam in $template_1$ wijzigt de context niet. Voor iedere nieuwe toegevoegde knoop k van $template_2$ geldt dat $context_2(k)$ gelijk is aan $context_1(k')$, waarbij k de kopie is van k'.

- s is een for-eachstatementknoop.

  Zij *expressie* de expressie horende bij s en *ouder* de ouder van s in $template_1$. Zij $c = (n, cp, cs, \alpha) := context_1(s)$ de context horende bij s. Zij $eval_{\wp}(expressie, t, \mathrm{c}) = \{n_1, n_2, \ldots, n_p\}$, waarbij $n_1, \ldots, n_p$ opgesomd zijn in pre-order van t. Zij nu *template* de template geworteld in het enige kind van s. *template* is van de vorm template(*rij*), waarbij *rij* de rij top-level deelbomen van *template* is. Creëer nu disjuncte kopieën $rij_1, \ldots, rij_p$ van *rij*.

  We kunnen nu $template_2$ definiëren als volgt: $template_2$ is bekomen uit $template_1$ door s te vervangen door de concatenatie van $rij_1, \ldots, rij_p$, waarbij elke top-level deelboom van rij_i, $i = 1, \ldots, p$, als ouder de knoop *ouder* heeft. De context voor de reeds bestaande knopen wijzigt niet. Voor iedere knoop k in $rij_i, i = 1, \ldots, p$, geldt: $context_2(k) = (n_i, i, p, \alpha)$ op voorwaarde dat k geen rechterbroer, of een afstammeling van een rechterbroer, is van een variabele-statementknoop uit rij_i en k geen afstammeling is van een for-eachstatementknoop.

Tot hier toe hebben we enkel de semantiek van controlestatementknopen beschreven. Nu willen we deze yield-functie bekijken als een subtree replacement system, zoals besproken in vorig hoofdstuk. We zullen dan later dit herschrijfsysteem gebruiken om de semantiek van de overige statementknopen te beschrijven. Het voordeel hiervan is dat we dan ook die herschrijvingen kunnen beschouwen als een subtree replacement system.

4.2.1 De yield-functie voor controlestatementknopen als een SRS

We gaan de yield-functie voor controlestatementknopen formeel definiëren als een subtree replacement system $(V, F, \rightarrow, \mathbf{R})$. V, de verzameling van alle labels, is hier de unie van het eindige alfabet Σ samen met alle labels die we gedefinieerd hebben in de abstracte syntax. Voorts breiden we deze verzameling nog uit door voor ieder label σ nog een extra verzameling labels te definiëren, namelijk het label bestaande uit σ en iedere mogelijke context. Merk op dat deze verzameling V oneindig kan zijn (door de variabelen

van het type result-tree-fragment). F is de verzameling van alle mogelijke bomen die syntactisch voldoen aan de definitie van een regel van een XSLT-programma. Tenslotte moeten we nog de functie $\rightarrow$ definiëren. Zij γ_1 en γ_2 twee configuraties over een inputboom t. Creëer nu volgende labelsubstitutie f: zij k een knoop uit een configuratie met label l. Indien de context van k gedefinieerd is, dan is het nieuwe label van k de string gevormd door l en de context van k. Is de context echter niet gedefinieerd, dan verandert het label niet. Stel nu dat s een controlestatement is uit γ_1, waarvan de context gedefinieerd is. We definiëren $\rightarrow$ als

$$\text{yield}(\gamma_1, s) = \gamma_2 \iff \text{f}(\text{template}(\gamma_1)) \rightarrow \text{f}(\text{template}(\gamma_2))$$

Laten we dit subtree replacement system mini-rewrite-system (mrs) noemen. Ook zullen we voortaan de notatie $\underset{\text{mrs}}{\rightarrow}$ gebruiken in plaats van $\rightarrow$, om duidelijk aan te geven dat het over een herschrijving van het mini-rewrite-system gaat. Wanneer we twee configuraties γ_1 en γ_2 hebben, schrijven we ook $\gamma_1 \underset{\text{mrs}}{\rightarrow} \gamma_2$ in plaats van echt de templates horende bij die configuraties te noteren.

We willen verder in de tekst aantonen dat ons mini-rewrite-system eindig is. Hiervoor bespreken we eerst een algemene techniek om terminatie te bewijzen, namelijk met behulp van multisets ordeningen. Dit deel is gebaseerd op de paper 'Proving Termination with Multisets Orderings' van Dershowitz en Manna [DM79].

definitie 4.10. Zij S een verzameling en $>$ een ordening op S. We noemen een set $(S, >)$ *welgevormd* als er geen oneindige dalende sequentie bestaat van elementen van S.

Voorbeeld 4.1. De set $(\mathbf{N}, >)$, waarbij $>$ de 'gewone' ordening is op natuurlijke getallen, is welgevormd. Voor ieder getal x zijn er maar eindig aantal getallen die strikt kleiner zijn dan x. Geen enkele sequentie kan namelijk onder 0 dalen. De set $(\mathbf{Z}, >)$ is echter niet welgevormd! ■

definitie 4.11. Een *partieel-geordende* set $(S, >)$ bestaat uit een verzameling S en een transitieve, irreflexieve binaire relatie $>$ op de elementen van S. Een dergelijke set noemen we *welgevormd* indien er geen oneindige dalende sequentie elementen bestaat van die set.

Voorbeeld 4.2. Zowel $(\mathbf{N}, >)$ als $(\mathbf{Z}, >)$, met $>$ weer de 'gewone' groter-dan relatie, zijn beide partieel-geordende sets. Echter is enkel de set $(\mathbf{N}, >)$ welgevormd. ■

definitie 4.12. Zij $(S, >)$ een partieel-geordende set. De *multisets over* S zijn alle eindige, ongeordende collecties van elementen van S waarin herhalingen van elementen mogen voorkomen. De verzameling van alle multisets over S noteren we met $\mathcal{M}(S)$.

Twee multisets A en B zijn gelijk, $A = B$, als en slechts als ieder element dat n keer voorkomt in A ook exact n keer voorkomt in B en, andersom, ieder element dat m keer voorkomt in B ook exact m keer voorkomt in A. De unie van A en B, $A \cup B$, bevat $n + m$ voorkomens van ieder element dat n keer voorkomt in A en m keer voorkomt in B.

Voorbeeld 4.3. De multisets $\{3, 3, 3, 4, 5, 5\}$ en $\{5, 4, 3, 5, 3, 3\}$ over de partieel-geordende set $(\mathbf{N}, >)$ zijn gelijk. De unie van de eerste multiset met de multiset $\{1, 6, 3, 4, 7\}$ is gelijk aan $\{3, 3, 3, 3, 6, 1, 7, 4, 4, 5, 5\}$. ■

definitie 4.13. Zij $(S, >)$ een partieel-geordende set. We definiëren de *multiset-ordening* $\gg$ *op* $\mathcal{M}(S)$ als:

$$M \gg M'$$

als er multisets $X, Y, Z \in \mathcal{M}(S)$ bestaan, met $X \neq \emptyset$, zodat

$$M = X \cup Z \text{ en } M' = Y \cup Z$$

en

$$(\forall y \in Y)(\exists x \in X)x > y$$

Deze definitie komt er op neer dat M' kleiner is dan M als en slechts als er uit M minstens één element verwijderd is (de multiset X) en vervangen door een eindig aantal (eventueel 0) elementen (de multiset Y), zodat ieder nieuw element kleiner is dan één van de verwijderde elementen.

lemma 4.1. Zij $(S, >)$ een partieel-geordende set en $\gg$ de multiset-ordening op $\mathcal{M}(S)$. Er geldt: $\gg$ is transitief-irreflexief.

Bewijs. Om aan te tonen dat $\gg$ irreflexief is, moeten we aantonen dat er geen multiset M over S bestaat zodat $M \gg M$. Dit doen we uit het ongerijmde: stel dat $M \gg M$. Er bestaat dan een niet-lege multiset X en twee multisets Y en Z zodat $M = X \cup Z$ en $M = Y \cup Z$ waarbij $(\forall y \in Y)(\exists x \in X)x > y$. Uiteraard geldt nu dat X en Y gelijk zijn en kunnen we dus schrijven: $(\forall y \in X)(\exists x \in X)x > y$. Dit laatste is echter onmogelijk, omdat X eindig is (er gaat namelijk geen getal groter zijn dan het grootste getal van X). Bijgevolg is onze aanname dat $\gg$ reflexief is fout, waardoor we weten dat $\gg$ irreflexief is.

Nu moeten we nog bewijzen dat $\gg$ transitief is. Beschouw hiervoor de irreflexieve relatie $\gg'$ op $\mathcal{M}(S)$: $Z \cup \{x\} \gg' Z \cup Y$ als $(\forall y \in Y)x > y$. Deze relatie vervangt één element x door één of meerdere elementen die allemaal kleiner zijn dan x. De multiset-ordening $\gg$ is de transitieve sluiting van $\gg'$, $M \gg M'$ als en slechts als M' van M afgeleid kan worden door het herhaaldelijk vervangen van één element. Hieruit volgt dat $\gg$ transitief is. □

Een laatste lemma dat we nodig hebben in het komende bewijs is het König's lemma. Hiervoor moeten we eerst nog een hulpconcept invoeren.

definitie 4.14. We noemen een boom t *eindig-vertakt* indien iedere knoop van t een eindig aantal kinderen heeft.

lemma 4.2 (König's lemma)**.** Iedere eindig-vertakte boom die een oneindig aantal knopen heeft, heeft een oneindig pad.

Bewijs. Zij t een eindig-vertakte boom met oneindig veel knopen. We noemen een knoop van t *goed* als het oneindig veel afstammelingen heeft. Omdat t oneindig veel knopen heeft, is de wortel van t zeker goed. Omdat t eindig-vertakt is, heeft ieder goede knoop n minstens één goed kind. Als dat niet zo was, dan zouden alle kinderen van n slechts een eindig aantal afstammelingen hebben. Dit zou echter impliceren dat n ook maar een eindig aantal afstammelingen heeft, wat een contradictie is met het gegeven dat n goed is. De wortel van t, r, die goed is, heeft dus zeker een kind r_1 dat eveneens goed is. Deze r_1 heeft ook weer een kind dat goed is, r_2. Op deze manier krijgen we dus een oneindige reeks $r, r_1, r_2, \ldots$, wat een oneindig pad vormt in t. □

stelling 4.1. De multiset-ordening $(\mathcal{M}(S), \gg)$ over $(S, >)$ is welgevormd als en slechts als $(S, >)$ welgevormd is.

Bewijs. Stel dat de multiset-ordening $(\mathcal{M}(S), \gg)$ over $(S, >)$ welgevormd is. We bewijzen uit het ongerijmde dat dan ook $(S, >)$ welgevormd is. Onze aanname is dus dat $(S, >)$ niet welgevormd is. Bijgevolg bestaat er een oneindige, dalende sequentie $s_1 > s_2 > s_3 > \ldots$ van elementen van S. De overeenkomstige sequentie van singletons $\{s_1\} \gg \{s_2\} \gg \{s_3\} \gg \ldots$ vormen een oneindige, dalende sequentie van elementen van $\mathcal{M}(S)$, waaruit blijkt dat $(\mathcal{M}(S), \gg)$ niet welgevormd is. Dit is echter tegenstrijdig met het gegeven, waardoor we moeten concluderen dat onze aanname fout was. Bijgevolg is $(S, >)$ welgevormd.

In de andere richting is het gegeven dat $(S, >)$ welgevormd is. We breiden de verzameling S uit met een nieuw element $\perp$, zodat $\perp$ kleiner is dan

alle andere elementen (met andere woorden $(\forall x \in S)(x \neq \bot) : x > \bot$). Hierdoor blijft S uiteraard nog steeds welgevormd. Stel nu, uit het ongerijmde, dat $(\mathcal{M}(S), \gg)$ over $(S, >)$ niet welgevormd is. Bijgevolg bestaat er een oneindige, dalende sequentie $M_1 \gg M_2 \gg M_3 \gg \ldots$ van multisets uit $\mathcal{M}(S)$. We leiden nu een contradictie af door het opbouwen van een boom. Iedere knoop in die boom is gelabeld met een element van S. De boom wordt als volgt opgebouwd: begin met een root te maken waarvan de kinderen overeenkomen met ieder element van M_1. Omdat $M_1 \gg M_2$ bestaan er multisets X, Y en Z zodat $M_1 = X \cup Z$ en $M_2 = Y \cup Z$, zodat X niet leeg is en $(\forall y \in Y)(\exists x \in X)x > y$. Voor iedere y van Y maak je een kind, met label y, aan de knoop van een overeenkomstige x (met andere woorden, een x die groter is dan y). Ieder element van X krijgt nog een extra kind met label $\bot$. Omdat X niet leeg is, zorgen de knopen met label $\bot$ ervoor dat de boom in iedere stap groeit, zelfs als Y leeg is. Omdat Y eindig is, hebben de knopen overeenkomstig met X steeds een eindig aantal kinderen. Herhaal nu dit proces voor $M_2 \gg M_3 \gg M_4 \gg \ldots$. In iedere stap van het proces wordt minstens één knoop toegevoegd aan de boom. Omdat de sequentie oneindig is, zal de boom ook oneindig groot worden. Ook weten we dat iedere knoop een eindig aantal kinderen heeft of, anders gezegd, de boom is eindig-vertakt. Bijgevolg kunnen we König's lemma gebruiken (lemma 4.2) om te concluderen dat de boom een oneindig pad bevat. Volgens de constructie van de boom zal ieder pad in de boom een dalende sequentie zijn volgens de ordening $>$ op S. Omdat deze ordening welgevormd is, zal die sequentie eindig zijn, wat dus een tegenstrijdigheid oplevert. Bijgevolg moet onze aanname, dat er een oneindige sequentie $M_1 \gg M_2 \gg M_3 \gg \ldots$ bestaat, fout zijn. Bijgevolg is $(\mathcal{M}(S), \gg)$ over $(S, >)$ welgevormd. □

lemma 4.3. Het mini rewrite systeem, zoals hierboven gedefinieerd, is eindig.

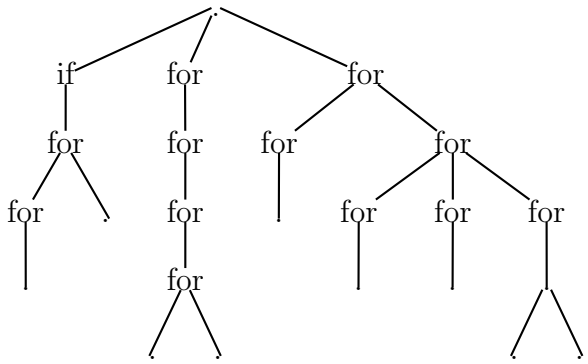
Bewijs. We weten reeds dat $(\mathbf{N}, >)$, met $>$ de 'gewone' groter-dan relatie op natuurlijke getallen, een partieel-geordende welgevormde set is. Uit stelling 4.1 weten we dan ook dat $(\mathcal{M}(\mathbf{N}), \gg)$ welgevormd is.

We beginnen met het definiëren van een terminatiefunctie die een configuratie van het mini-rewrite-system mapt op een multiset over $(\mathbf{N}, >)$.

We noemen een for-eachstatementknoop een *top-for* indien deze knoop geen afstammeling is van een for-eachstatementknoop. De *nestingsdiepte* van een dergelijke knoop is het maximaal aantal geneste for-eachstatementknopen.

We creëren nu een multiset m_γ voor een configuratie γ als volgt:

1. Initialiseer m_γ als de lege multiset $\{\}$
2. Voeg voor iedere top-for zijn nestingsdiepte toe aan m_γ



Figuur 4.2: Voorbeeld van de terminatiefunctie

3. Voeg evenveel nullen toe aan m_y als er knopen zijn in template(γ), die niet gelabeld zijn met 'template'.

De opbouw van het bovenstaande multiset is dus de beschrijving van een terminatiefunctie die een configuratie afbeeldt op een multiset.

Voorbeeld 4.4. Beschouw de boom gegeven in figuur 4.2. Stel dat deze boom de template is van een configuratie γ. De knopen gelabeld met '.' zijn van geen belang, maar zijn zeker geen for-eachstatementknopen. Omdat we knopen met label 'template' niet tellen, zijn deze ook niet getekend. De multiset m_γ over $(\mathbf{N}, >)$ horende bij deze configuratie is gelijk aan $\{2, 4, 3, 0^{24}\}$. Hierbij staat 0^n voor een sequentie van n nullen. ■

Zij γ_1 een configuratie waarin we een controlestatementknoop k, waarvan de context gedefinieerd is, wegwerken. De resulterende configuratie noemen we γ_2. Zij m_1 en m_2 de bijbehorende multisets over $(\mathbf{N}, >)$. We hebben nu een aantal mogelijkheden. Waneer k een if-thenstatementknoop is, zal door de uitwerking het aantal knopen van template(γ) zeker dalen. Als de test horende bij k true oplevert, is er één knoop minder, namelijk de knoop k zelf. In het andere geval daalt het aantal knopen met minstens één omdat niet alleen k verdwijnt, maar ook de template horende bij die knoop. In het geval dat k een choose-when-otherwise-statementknoop is, kunnen we een analoge redenering maken. Het aantal knopen vermindert dus in alle gevallen. Bijgevolg zal de multiset m_2 minstens één 0 minder bevatten. Volgens de definitie van $\gg$ is dan $m_1 \gg m_2$.

Een andere mogelijkheid is dat k een for-eachstatementknoop is. Wanneer de expressie e horende bij k geen knopen selecteert, zal het aantal knopen dalen: de knoop k en alle afstammelingen van k verdwijnen. Ook zal de

nesting d van k uit de multiset verdwijnen. Er gebeuren dus een aantal verwijderingen uit de multiset m_1, zonder dat er vervangingen volgen. Bijgevolg is $m_1 \gg m_2$. Hetzelfde verhaal gebeurt wanneer de template horende bij k leeg is.

Wanneer er exact één knoop geselecteerd werd door e, zal template(γ_2) knoop k niet meer bevatten en is dus het aantal knopen al zeker gedaald, waardoor er minder knoopnullen zullen zijn. De nestingsdiepte van k wordt ook verwijderd. Het kan zijn dat de template horende bij k weer for-each-statementknopen bevat. De nestingsdieptes van de nieuwe top-fors worden toegevoegd aan de multiset. Deze toevoegingen zijn strikt kleiner dan d, waardoor deze vervangingen zeker kleiner zijn dan een waarde die verwijderd werd (namelijk d zelf). We mogen dus besluiten dat $m_1 \gg m_2$.

Het ingewikkeldste geval is echter wanneer e minstens twee knopen selecteert en de template horende bij k niet leeg is. In de praktijk zal dit geval het meest voorkomen. Het aantal knopen is dan zeker niet gedaald. Er kunnen dus een aantal nullen toegevoegd zijn aan de multiset. Echter is de diepte d minstens 1 en wordt zeker verwijderd. De toevoeging van nullen vormt dus geen probleem, omdat 0 zeker kleiner is dan een verwijderd getal. Wanneer d strikt groter is dan 1, zal er een reeks nieuwe nestingdieptes toegevoegd worden. De waardes van deze nestingsdieptes zijn allemaal strikt kleiner dan een verwijderd getal (d). Bijgevolg geldt ook hier dat $m_1 \gg m_2$.

In iedere stap van het mini-rewrite-system is de multiset gevormd uit de nieuwe configuratie strikt kleiner dan de multiset horende bij de gegeven configuratie. Bijgevolg hebben we een terminatiefunctie, die een configuratie mapt op een multiset over $(\mathbf{N}, >)$, waarvan de waarde daalt in iedere stap. Omdat we weten dat $(\mathcal{M}(\mathbf{N}), \gg)$ welgevormd is, mogen we besluiten dat ons mini-rewrite-system eindig is. □

Voorbeeld 4.5. Beschouw figuur 4.3. In stap 1 is de multiset horende bij de template gelijk aan $\{1, 3, 0^{10}\}$. We werken de for-eachstatementknoop uit, die blijkbaar 2 knopen selecteert. De resulterende template, in stap 2 getekend, heeft als multiset $\{1, 2, 2, 0^{13}\}$. Uit de multiset zijn een reeks nullen verwijderd, samen met de nestingsdiepte 3. We voegden een nieuwe reeks nullen toe en twee keer de nestingsdiepte 2. We zien dat alle toevoegingen kleiner zijn dan een verwijderd element. In de volgende stap werken we de if-thenstatementknoop weg, waarvan de test blijkbaar false oplevert. Bijgevolg verdwijnen er een reeks nullen, maar ook een nestingsdiepte, namelijk 1. We voegen geen nieuwe knopen toe. De resulterende multiset is $\{2, 2, 0^9\}$. In de volgende stap blijkt dat een expressie bij een for-lus geen knopen selecteert. Het gevolg daarvan is dat een volledige subtree verdwijnt. De multiset die daaruit volgt is $\{2, 0^5\}$. De expressie bij de laatste top-for selecteert maar één

knoop. Er verdwijnt dus één knoop zonder dat er nieuwe knopen toegevoegd worden. De multiset horende bij deze stap is $\{1, 0^4\}$. Als laatste stap wordt de overblijvende for-eachstatementknoop uitgewerkt, waarvan de expressie drie knopen selecteert. De multiset is $\{0^7\}$. Merk op dat in de laatste stap de multiset altijd een sequentie nullen is. ■

lemma 4.4. Het mini rewrite systeem, zoals hierboven gedefinieerd, is locaal confluent.

Bewijs. Zij u, v, w configuraties en zij k en l twee controle-statementknopen in template(u) waarvan de context gedefinieerd is. We nemen aan dat we k in de herschrijfstap $u \to v$ wegwerken, in de stap $u \to w$ elimineren we l.

Het eerste geval dat we beschouwen is wanneer k en l onafhankelijk zijn, met andere woorden k is geen voorouder van l en l is geen voorouder van k. Een herschrijfregel gaat een template enkel en alleen aanpassen in één knoop. Omdat l geen descendant van k is, komt l ongewijzigd voor in template(v). Hetzelfde geldt voor k. Omdat k geen descendant is van l, komt k ongewijzigd voor in template(w). Bijgevolg kunnen we in template(v) l op dezelfde manier herschrijven als in $u \to w$, we noteren dat resultaat als z. In template(w) herschrijven we k zoals k getransformeerd werd in $u \to v$. Omdat k en l onafhankelijk zijn, resulteert deze herschrijving ook in z. Bijgevolg is locale confluentie aangetoond.

We bespreken nu het geval dat k een voorouder is van l. Het geval dat l een voorouder is van k is analoog. We hebben een aantal mogelijkheden voor de knoop k. Als eerste stellen we dat k een for-eachstatementknoop is. Dit is een echter een onmogelijke situatie! Je kan l nooit herschrijven alvorens je k herschrijft omdat l geen context heeft! (Afstammelingen van een for-eachstatementknoop krijgen nooit een context.)

Wanneer k een if-thenstatementknoop is, kunnen we een verdere opsplitsing maken afhankelijk van het resultaat van de test. Stel dat k een test bevat die tot false geëvalueerd wordt. template(v) is dan gelijk aan template(u), behalve dat in template(v) de knoop k verwijderd is. Stel dan z gelijk aan v. Door eerst l te herschrijven in template(u), is template(w) gelijk aan template(u), met uitzondering dat knoop k een gewijzigde template als kind heeft. Pas nu de herschrijfregel voor k op w toe. Net als voorheen zal k verwijderd worden uit template(w), waardoor we template(u) krijgen met daaruit k verwijderd. Maar deze configuratie is dan hetzelfde als z, waardoor locale confluentie bewezen is. In het andere geval heeft k een test die tot true geëvalueerd wordt. Zij dan t de template geworteld in het unieke kind van k. t heeft als afstammeling l. Omdat alle afstammelingen van t voorkomen in template(v), komt l ook voor in template(v). We kunnen dan de herschrijfregel op l toepassen, het resultaat noemen we z. Wanneer we in

u echter eerst l herschreven, kregen we een configuratie w. template(w) bevat nog steeds k, hoewel er afstammelingen van k gewijzigd zijn. k heeft nu niet langer een afstammeling l, maar wel een afstammeling die het resultaat is van l te herschrijven. Door nu de if-then-regel toe te passen in k, krijgen we z. Hiermee is locale confluentie aangetoond.

Tenslotte kan het nog zijn dat k een choose-when-otherwisestatement-knoop is. Stel dat k geen enkele test bevat die tot true geëvalueerd kan worden en er is geen 'otherwise'-template. v is dan gelijk aan u, behalve dat in template(v) de knoop k verwijderd is. Stel nu z gelijk aan v. Wanneer we eerst volgens l herschrijven, is template(w) gelijk aan de template(u), behalve dat een template geworteld in k gewijzigd is. Als we nu op w de herschrijfregel voor k toepassen, krijgen we een configuratie c die gelijk is aan w, behalve dat in template(c) de knoop k verwijderd is. Deze resulterende configuratie is gelijk aan z. Wanneer k een test bevat die tot true geëvalueerd wordt, of k bevat een 'otherwise'-template, is er een template t die knoop k zal vervangen. We nemen eerst aan dat l geen knoop is van t. Het resultaat van u te herschrijven volgens de regel voor k, levert configuratie v. De template van v bevat knoop l niet meer. We stellen z gelijk aan v. Indien we in u eerst l herschreven, werd één kind van k herschreven. De resulterende configuratie is w. Door nu k in w uit te werken, vervangen we k eveneens door template t. Omdat l geen knoop van t was, komt l ook niet voor in de resulterende configuratie. Het resultaat is dus gelijk aan z. Stel nu echter dat t wel de knoop l bevat. Omdat alle afstammelingen van t voorkomen in template(v), komt l ook voor in template(v). We kunnen dan de herschrijfregel op l toepassen, het resultaat noemen we z. Wanneer we in u echter eerst l herschreven, kregen we een configuratie w. template(w) bevat nog steeds k, hoewel een kind van k gewijzigd zijn. Door nu k uit te werken, krijgen we als nieuwe template die template waarin k vervangen is door t', waarbij t' gelijk is aan t met daarin l herschreven. Deze configuratie is gelijk aan z.

We zien dus dat we de configuraties v en w steeds kunnen herschrijven naar éénzelfde configuratie z. Hieruit volgt dat ons mini-rewrite-system locaal confluent is. □

lemma 4.5. Het mini-rewrite-systeem, zoals hierboven gedefinieerd, is confluent.

Bewijs. Uit lemma 4.3 weten we dat het mini-rewrite-system eindig is. Het vorige lemma toonde locale confluentie aan. Door toepassing van Newman's lemma (lemma 3.1), volgt onmiddellijk dat het mini-rewrite-systeem confluent is. □

We gaan nu verder met de semantiek voor de overige statementknopen.

definitie 4.15. Zij t een boom. Zij $\gamma_1 = (template_1, context_1)$ een configuratie over t en zij s een niet-controlestatementknoop van $template_1$, zodanig dat $context_1(s)$ gedefinieerd is. We definiëren een nieuwe configuratie $\gamma_2 = (template_2, context_2)$ zodanig dat $\text{step}(\gamma_1, s) = \gamma_2$ als volgt:

- s is een apply-templatesstatementknoop. De enige kinderen zijn with-paramstatementknopen. We eisen dat de templates van deze kinderen terminaal zijn.

  Zij *ouder* de ouder van s in $template_1$; e de expressie horende bij s; *mode* de mode van s; en $c{=}(n, cp, cs, \alpha) := context_1(s)$. We definiëren eerst een nieuwe variabeletoekenning $\bar{\alpha}$ als volgt: voor iedere $v \in V_g$ geldt $\bar{\alpha}(v) = \alpha(v)$. Voor de variabelen in V_l stellen we:

  - Als v niet gedefinieerd wordt door s, dan is $\bar{\alpha}(v)$ ook niet gedefinieerd ($\bar{\alpha}(v)$ is *wel* gedefinieerd als $v \in V_g$).
  - Als v gedefinieerd wordt door s en $type(v)$ = 'teller' of 'node-set', zij dan *expr* de expressie behorende bij de definitie van v door s. Er geldt $\bar{\alpha}(v) = eval_{\wp}(expr, t, c)$.
  - Als v gedefinieerd wordt door s en $type(v)$ = 'result-tree-fragment', zij dan *kind* het kind van s dat het with-paramstatement van v voorstelt. We weten dat *kind* een terminale template *temp* als kind heeft. *temp* is van de vorm template(*rij*), waarbij *rij* de concatenatie is van de top-level deelbomen van *temp*. $\bar{\alpha}(v) = rij$.

  Zij $eval_{\wp}(e, t, \text{c}) = \{n_1, \ldots, n_p\}$, waarbij $n_1, \ldots, n_p$ opgesomd zijn in pre-order van t. We zoeken voor elke $n_i, i \in \{1, \ldots, p\}$, de eerste regel $(pattern_i, mode_i, template_i)$ in Π waarbij

  - $mode_i = mode$
  - $n_i \in eval_{\wp}(pattern_i, t, \text{c})$

  Indien nu voor een zekere i tussen 1 en p geen regel gevonden kon worden, creëren we zelf een template, $template_i$. Dit is een boom waarvan de root gelabeld is met 'template'. Het enige kind van deze root is een apply-templatesstatementknoop met als label 'apply-templates', samen met de expressie 'children'. Het laatste onderdeel van het label geeft de mode aan, deze is *mode*.

  Creëer nu disjuncte kopieën $template'_1, \ldots, template'_p$ van $template_1$, $\ldots$, $template_p$. Elke template $template'_i$ is van de vorm template(rij_i),

waarbij rij_i de concatenatie is van de top-level deelbomen van $template'_i$. Zij *nieuw* de concatenatie van $rij_1, \ldots, rij_p$.

We maken nu een tussentijdse configuratie $\gamma = (template, context)$. *template* is gedefinieerd als volgt: vervang knoop s door *nieuw*, waarbij elke top-level deelboom van *nieuw* als ouder de knoop *ouder* heeft. De contextfunctie voor γ, *context*, is omschreven als volgt: de context van een knoop die reeds in template(γ_1) zat, blijft hetzelfde. Zij k echter een nieuwe toegevoegde knoop uit rij_i (k is dan ook een knoop van *nieuw*). Indien k een right-sibling, of een afstammeling van een right-sibling, is van een variabelestatement in rij_i, dan is de context voor k niet gedefinieerd. In het andere geval is de context van k gelijk aan $(n_i, i, p, \bar{\alpha})$, op voorwaarde dat k geen afstammeling is van een for-eachstatementknoop.

We definiëren nu γ_2 als $\gamma \xrightarrow[\text{mrs}]{*} \gamma_2$, zodat γ_2 geen controlestatementknopen meer bevat waarvan de context gedefinieerd is. Wegens lemma 4.3 weten we dat deze herschrijving slechts een eindig aantal stappen vergt en verder door lemma 4.5 dat het resultaat van deze herschrijving uniek is.

- s is een call-templatestatementknoop. De enige kinderen zijn with-paramstatementknopen van het type 'result-tree-fragment'. We eisen dat de templates van deze kinderen terminaal zijn.

  Zij *ouder* de ouder van s in $template_1$, *naam* de naam van s en $c = (n, cp, cs, \alpha) :=context_1(s)$ de context horende bij s. We beginnen met het definiëren van een nieuwe contextfunctie $\bar{\alpha}$. We stellen voor iedere $v \in V_g$ dat $\bar{\alpha}(v) = \alpha(v)$. Voor iedere v in V_l geldt:

  - Als v niet gedefinieerd wordt door s, dan is $\bar{\alpha}(v)$ niet gedefinieerd ($\bar{\alpha}(v)$ is *wel* gedefinieerd als $v \in V_g$).
  - Als v gedefinieerd wordt door s en *type*(v) = 'teller' of 'node-set', zij dan *expr* de expressie behorende bij de definitie van v door s. Er geldt $\bar{\alpha}(v) = eval_{\wp}(expr, t, \text{c})$.
  - Als v gedefinieerd wordt door s en *type*(v) = 'result-tree-fragment', zij dan *kind* het kind van s dat het with-paramstatement van v voorstelt. We weten dat *kind* een terminale template *temp* als kind heeft. *temp* is van de vorm template(*rij*), waarbij *rij* de concatenatie is van de top-level deelbomen van *temp*. $\bar{\alpha}(v) = rij$.

  We zoeken nu de (enige) regel r met naam *naam*. Zij *template* de bij r horende template. Zij *template*$'$ een kopie van *template*. Hierbij is

template$'$ van de vorm template(*rij*), waarbij *rij* de concatenatie is van de top-level deelbomen van *template*$'$.

We maken nu een tussentijdse configuratie $\gamma = (template, context)$. *template* definiëren we als volgt: vervang knoop s door rij, waarbij elke top-level deelboom van *rij* als ouder de knoop *ouder* heeft. De contextfunctie voor γ, *context*, is omschreven als volgt: de context van een knoop die reeds in template(γ_1) zat, blijft hetzelfde. Zij k echter een nieuwe toegevoegde knoop uit *rij*. Indien k een right-sibling, of een afstammeling van een right-sibling, is van een variabelestatement in de deelboom *ouder*, dan is de context voor k niet gedefinieerd. In het andere geval is de context van k gelijk aan $(n, cp, cs, \bar{\alpha})$, op voorwaarde dat k geen afstammeling is van een for-eachstatementknoop.

We definiëren nu γ_2 als $\gamma \xrightarrow[\text{mrs}]{*} \gamma_2$, zodat γ_2 geen controlestatementknopen meer bevat waarvan de context gedefinieerd is. Wegens lemma 4.3 weten we dat deze herschrijving slechts een eindig aantal stappen vergt en verder door lemma 4.5 dat het resultaat van deze herschrijving uniek is.

- s is een variabelestatementknoop. Indien s een variabele van het type 'result-tree-fragment' definieert, eisen we dat het enige kind van s een terminale template is.

  Zij *ouder* de ouder van s, *naam* de naam van de variabele gedeclareerd door s en de context horende bij s is gelijk aan $c = (n, cp, cs, \alpha)$:=$context_1(s)$. We creëren nu een nieuwe variabeletoekenning $\bar{\alpha}$. Voor iedere $v \in V$ met $v \neq naam$ geldt $\bar{\alpha}(\mathrm{v}) = \alpha(v)$. We hebben nu nog twee mogelijkheden. Enerzijds kan s een variabele declareren van het type 'node-set' of 'teller'. s heeft dan geen kinderen. We stellen r gelijk aan de expressie van het statement en $\bar{\alpha}(naam) = eval_{\wp}(r, t, c)$. Anderzijds kan s een variabelestatement zijn voor een variabele van het type 'result-tree-fragment'. Zij dan *kind* het enige kind van s. *kind* is dan van de vorm template(*rij*) waarbij *rij* de concatenatie is van de top-level deelbomen van *kind* is. We stellen $\bar{\alpha}(naam)$ gelijk aan *rij*.

  We maken nu de configuratie γ_2. $template_2$ stellen we gelijk aan $template_1$ met daaruit de deelboom geworteld in s verwijderd. We splitsen voor het bepalen van de contextfunctie $template_2$ op in twee delen. Zij *rechts* het forest van bomen waarvan de root een knoop is die overeenkomstig was met een right-sibling van s en die een linker-sibling is van de eerste right-sibling van s waarvoor de context gedefinieerd is. Al de knopen in *rechts* hebben nog geen context. Voor iedere knoop k in $template_2$

die niet voorkomt in *rechts* stellen we $context_2(k) = context_1(k)$. Voor iedere knoop k in *rechts* die een right-sibling, of een afstammeling van een right-sibling, is van een variabelestatement, is de context niet gedefinieerd. Voor iedere andere knoop k in *rechts* geldt $context_2(k) = (n, cp, cs, \bar{\alpha})$, op voorwaarde k geen afstammeling is van een for-each-statementknoop.

- s is een copy-ofstatement.

  Zij *ouder* de ouder van s in $template_1$ en c gelijk aan $context_1(s)$. Enerzijds kan s een expressie e van het type node-set in zijn label hebben staan. Zij dan $\{n_1, \ldots, n_p\}$ de node-set bekomen door het uitvoeren van $\text{eval}_\wp(e, t, c)$, waarbij $n_1, \ldots, n_p$ opgesomd zijn in pre-order van t. Creëer nu disjuncte kopieën van $n_1, \ldots, n_p$ en zij *rij* de concatenatie van deze kopieën. Anderzijds kan s in zijn label een variabele v van het type 'result-tree-fragment' hebben staan. Zij dan *rij* een kopie van $\alpha(v)$. We creëren nu $template_2$ als volgt: $template_2$ is bekomen door in $template_1$ de knoop s te vervangen door *rij*. Iedere top-level deelboom van *rij* heeft dan als ouder de knoop *ouder*. Voor iedere knoop k die voorkwam in $template_1$ definiëren we $context_2(k)$=$context_1(k)$. De context van iedere nieuwe toegevoegde knoop (met andere woorden iedere knoop van *rij*) is c.

4.2.2 De step-functie voor niet-controlestatementknopen als een SRS

Net zoals de yield-functie, willen we de step-functie voor het uitwerken van statementknopen beschrijven als een subtree replacement system. We beginnen met het definiëren van dit SRS $\Re$. Deze is volledig hetzelfde als ons mini-rewrite-system (zie pg. 66), behalve dat de herschrijfrelatie $\rightarrow$ nu gedefinieerd is met de step-functie voor niet-controlestatementknopen. Er geldt dus:

Stel dat s een niet-controlestatementknoop is uit γ_1, waarvan de context gedefinieerd is. We definiëren $\rightarrow$ als

$$\text{step}(\gamma_1, s) = \gamma_2 \iff \text{f}(\text{template}(\gamma_1)) \rightarrow \text{f}(\text{template}(\gamma_2))$$

We willen nu, net als het mini-rewrite-systeem, aantonen dat $\Re$ confluent is. Omdat het eenvoudig is om een oneindige herschrijving te maken (zie ook hoofdstuk 5 over terminatie), is Newman's lemma hier niet toepasbaar. We zullen daarom stelling 3.4 uit hoofdstuk 3 gebruiken. De eerste eis van die stelling is dat $\Re$ unequivocal moet zijn. Dit is hier zeker. Of $\Re$ gesloten is, is op het eerste zicht niet duidelijk.

lemma 4.6. Het subtree replacement system $\Re$, zoals hierboven gespecificeerd, is gesloten.

Bewijs. Er zijn vier categoriën van herschrijfregels in het $\Re$, namelijk die voor het omvormen van een apply-templatesstatementknoop, van een call-templatestatementknoop, van een variabelestatementknoop en tenslotte van een copy-ofstatementknoop. We zullen nu voor elk van deze vier types argumenteren hoe we een residumap kunnen toekennen aan die regels, zodat aan de voorwaarden van de definitie van gesloten zijn voldaan wordt.

- regels voor een apply-templatesstatementknoop

  Beschouw de transitie $\phi_0 \rightarrow \psi_0$ uit figuur 4.4. We maken nu een residumap voor deze transitie. Voor iedere knoop n in 'links' of 'rechts' is $r[\phi_0, \psi_0](n)$ de overeenkomstige knoop in ψ_0. Voor de overige knopen stellen we de residumap gelijk aan $\emptyset$. Dit vormt geen enkel probleem, wat we nu zullen argumenteren.

  We moeten aantonen dat $\phi_1 \rightarrow \psi_1$ een regel is in binnen $\Re$, waarbij ϕ_1 de boom is waarin we een knoop n van ϕ_0 herschrijven. ψ_1 is de boom ψ_0 waarin we iedere knoop in de residumap van n herschrijven met behulp van dezelfde regel waarmee n herschreven werd. We gaan nu alle mogelijke knopen af en beginnen met diegene die niet in 'links' of 'rechts' zitten.

  De apply-templatesstatementknoop heeft een lege verzameling als residumap. Dit vormt op zich geen probleem omdat we deze knoop in de definitie van gesloten zijn toch niet moeten beschouwen. Dit omdat in die definitie vermeld staat dat er een regel $\phi \rightarrow \psi$ moet bestaan die toepasbaar is in de beschouwde knoop. Een dergelijke regel komt duidelijk niet voor in $\Re$. Er is simpelweg geen herschrijfregel $\phi \rightarrow \psi$ waarbij de root van ϕ gelabeld is met 'apply-templates'. Deze redenering kunnen we ook maken voor de knopen gelabeld met 'with-param' en alle knopen onder de template van deze with-paramstatementknopen omdat deze templates terminaal zijn: er zijn geen regels $\phi \rightarrow \psi$ in $\Re$ waarvan de root van ϕ het label 'with-param' draagt of waarbij ϕ een boom is die enkel uit constante knopen bestaat. Beschouw nu nog de knopen gelabeld met template onder de with-paramstatementknopen. Er is in $\Re$ geen herschrijfregel die een terminale template gaat vervangen. Bijgevolg moeten we die knopen ook niet beschouwen volgens de definitie van gesloten zijn.

  Beschouw nu een knoop n uit 'links' waarvoor een herschrijfregel $\phi \rightarrow \psi$ bestaat zodat $\phi_0/n = \phi$. We weten dat $r[\phi_0, \psi_0](n)$ één knoop bevat,

k. De deelbomen geworteld in n en k zijn volledig gelijk. We kunnen dus ϕ_1 construeren door $\phi \rightarrow \psi$ toe te passen in n. ψ_1 is het resultaat van de beschouwde herschrijfregel toe te passen in k. We moeten nu nog argumenteren dat $\phi_1 \rightarrow \psi_1$ een regel is in $\Re$.

De herschrijving van de apply-templates heeft enkel invloed op de deelboom geworteld in de apply-templatesstatementknoop. Ook wordt tijdens deze herschrijving geen gebruik gemaakt van knopen die onafhankelijk zijn van deze knoop. Iedere knoop in 'links' is duidelijk onafhankelijk van de apply-templatesstatementsknoop. Bijgevolg kunnen we de apply-templates op dezelfde manier herschrijven als in $\phi_0 \rightarrow \psi_0$. We kunnen dus besluiten dat $\phi_1 \rightarrow \psi_1$ een regel is binnen $\Re$.

Indien we een knoop n uit links kiezen waarvoor geen herschrijfregel $\phi \rightarrow \psi$ bestaat zodat $\phi_0/n = \phi$, moeten we daar geen rekening mee houden volgens de definitie van gesloten zijn. Deze knopen kunnen dus geen aanleiding zijn tot het niet gesloten zijn van $\Re$.

We kunnen nu dezelfde argumentatie gebruiken voor iedere knoop n in 'rechts'.

De enige knoop die we nu nog niet besproken hebben, is de root van ϕ_0. Dit is echter geen probleem omdat deze buiten beschouwing mag gelaten worden volgens de definitie van gesloten zijn. Bijgevolg is voor iedere herschrijfregel die een apply-templates herschrijft bewezen dat we een residumap kunnen toekennen met de eigenschappen gegeven in de definitie van gesloten zijn.

- regels voor het omvormen van een call-templatestatementknoop

  Dit is volledig analoog als het vorige onderdeel.

- regels voor het omvormen van een variabelestatementknoop

  Beschouw ϕ_0 en ψ_0 uit figuur 4.5. Net zoals bij apply-templates is de residumap voor een knoop uit 'links' van ϕ_0 de overeenkomstige knoop in 'links' van ψ_0. We kunnen dezelfde argumentatie als voorheen gebruiken dat dit geen tegenstrijdigheden oplevert met de definitie van gesloten zijn. Het rechtergedeelte is opgesplitst in twee delen. rechts_1 zijn de knopen die nog geen context hebben, rechts_2 de knopen die een context kunnen hebben. De eerste knoop van rechts_2 heeft zeker een context. Deze opdeling komt dus overeen met het rechtergedeelte zoals uitgelegd in de semantiek van een variabelestatementknoop. Voor de knopen in rechts_2 stellen we de residumap gelijk aan de overeenkomstige knoop. Dit vormt geen tegenstrijdigheden, zoals hierboven reeds

werd uitgelegd. Voor de overige knopen in ϕ_0, waaronder dus rechts_1, geldt dat de residumap leeg is. Dit vormt geen tegenstrijdigheden in definitie van gesloten zijn, omdat er voor al deze knopen *geen* regel $\phi \to \psi$ bestaat die toegepast kan worden. Voor iedere knoop in rechts_1 is dit vrij triviaal omdat er geëist wordt bij een herschrijving dat de knoop een context heeft. Dit is in rechts_1 nooit het geval, omdat rechts_1 enkel knopen bevat die geen context hebben. Voor de variabelestatementknoop is er geen herschrijfregel. Ook is er geen herschrijfregel voor terminale templates. De wortel van ϕ_0 moet niet beschouwd worden volgens de definitie van gesloten zijn. We mogen dus ook voor de herschrijfregels van variabelestatementknopen besluiten dat we steeds een residumap aan de knopen in ϕ_0 kunnen toekennen, overeenkomstig aan de eigenschappen beschreven in de definitie van gesloten zijn.

- regels voor het omvormen van een copy-of statementknoop

  In figuur 4.6 staat de situatie getekend voor copy-ofstatementknopen. Voor iedere knoop in links of rechts in ϕ_0 is de residumap de overeenkomstige knoop in ψ_0. De residumap van de copy-ofstatementknoop is leeg. De argumentatie dat deze toekenning voldoet aan de eigenschappen van gesloten zijn, is volledig analoog aan diegene besproken bij andere types statementknopen.

Aan het tweede onderdeel van gesloten zijn, dat de residumappen van twee onafhankelijke knopen onafhankelijk zijn, is voldaan. We mappen veel knopen op de lege verzameling, dus daar kunnen al zeker geen tegenstrijdigheden met de tweede voorwaarde voldaan. Andere knopen worden altijd gemapt op de overeenkomstige knoop, waar de ouder-van relatie nergens gewijzigd is. Dit vormt dus ook geen tegenstrijdigheid.

We kunnen dus besluiten dat we een residumap kunnen toekennen aan iedere herschrijfregel $\phi_0 \to \psi_0$ in $\mathfrak{R}$, die voldoet aan de eigenschappen gespecificeerd in de definitie van gesloten zijn. Bijgevolg is ons SRS gesloten. □

stelling 4.2. Het SRS $\mathfrak{R}$, dat we hierboven specificeerden, is confluent.

Bewijs. Doordat $\mathfrak{R}$ unequivocal is en het gebruik van lemma 4.6, kunnen we stelling 3.4 toepassen. Hieruit volgt dus onmiddellijk dat $\mathfrak{R}$ confluent is. □

We hebben nu de semantiek van alle mogelijke statementknopen beschreven. We moeten nu nog definiëren hoe een programma start op een gegeven inputboom.

definitie 4.16. Zij t een boom en zij $\Pi = (V = V_g \cup V_l, \mathit{type}, R, <, <_g, \mathit{def})$ een XSLT-programma. We definiëren de run van Π op boom t als volgt:

1. Zij $v_1, v_2, \ldots, v_k$ de verschillende globale variabelen, zodanig dat $v_i <_g v_{i+1}$ voor iedere $i = 0, \ldots, k-1$. Zij r de root van t. Zij u het variabelestatement horende bij v_1, gedefinieerd door *def* (m.a.w. $def(v_1) = u$). Indien v_1 een variabele is van type node-set of teller, zij dan *expressie* de expressie horende bij u. We maken nu een variabeletoekenning α_1, zodat $\alpha_1(v_1) = \text{eval}(expressie, t, c)$, waarbij c de context $(r, 1, 1, \emptyset)$[3] is. Is v_1 echter een variabele van het type result-tree-fragment, creëer dan een nieuwe regel binnen het programma. Deze regel heeft als naam een naam die nog niet voorkomt, bijvoorbeeld 'berekenVariabelev1'. De template van deze regel is het unieke kind horende bij het variabelestatement u. Maak nu een configuratie $\gamma_1 = (template_1, context_1)$, waarbij $template_1$ de template is bestaande uit één knoop, namelijk een call-templatestatementknoop met als label 'call-template berekenVariabelev1'. De context horende bij deze knoop is het tupel $(r, 1, 1, \emptyset)$.

   Zij nu γ_1' de terminale configuratie die volgt uit de uitwerking van γ_1. Omdat we weten dat XSLT 1.0 confluent is, is γ_1' uniek bepaald. De template van γ_1' is van de vorm template(rij). We stellen nu: $\alpha_1(v_1) = rij$. We eindigen met de regel met naam 'berekenVariabelev1' te verwijderen uit de definitie van het programma.

   We passen nu dezelfde strategie toe op de volgende variabelen. Zij $i = 2, \ldots, k$. Zij u_i het variabelestatement horende bij v_i. We creëren de variabeletoekenning α_i. Voor iedere variabele $v_j \in V_g$, $v_j <_g v_i$, geldt $\alpha_i(v_j) = \alpha_{i-1}(v_j)$. Als v_i een variabele is van het type node-set of teller, zij dan *expressie* de overeenkomstige expressie. We stellen $\alpha_i(v)$ gelijk aan $\text{eval}(expressie, t, c)$, waarbij c de context $(r, 1, 1, \alpha_{i-1})$ is. In het andere geval, wanneer v_i een variabele van het type result-tree-fragment is, maken we een nieuwe regel met een naam die niet voorkomt in de programmadefinitie, bijvoorbeeld 'berekenVariabelevi'. We maken nu de configuratie $\gamma_i = (template_i, context_i)$. Hierbij is $template_i$ de template bestaande uit één knoop, een call-templatestatementknoop, met als label 'call-template berekenVariabelevi'. $(r, 1, 1, \alpha_{i-1})$ is de context van deze knoop.

   Zij nu γ_i' de terminale configuratie die volgt uit de uitwerking van γ_i. Omdat we weten dat XSLT 1.0 confluent is, is γ_i' uniek bepaald. De template van γ_i' is van de vorm template(rij). We stellen nu: $\alpha_i(v) = rij$ en verwijderen de zopas gecreëerde regel uit ons programma.

   We zijn dus in staat om een variabeletoekenning α_k af te leiden.

[3] Hierbij stelt $\emptyset$ de lege functie voor.

2. Merk op dat we tot nu toe enkel nog maar de globale variabelen hebben berekend. De start van de transformatie start eigenlijk nu pas. Hiervoor maken we een template *template*. De root van deze boom, gelabeld met 'template', heeft 1 kind. Dit kind heeft als label 'apply-templates', de expressie 'root' uit $\wp$ en tenslotte de mode 'start'. Maak nu een configuratie γ_s bestaande uit de template *template* en de contextfunctie die aan iedere knoop van *template* de context $(r, 1, 1, \alpha_k)$ toekent.

3. Zij γ_e de unieke configuratie bekomen door de uitwerking van γ_s. De template van deze configuratie is terminaal en van de vorm template(rij). We definiëren de output van XSLT-programma Π op inputboom t als rij.

4.2.3 Voorbeeld van een run

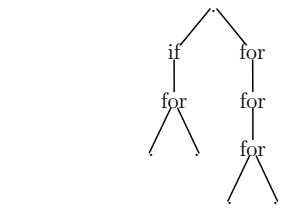
In sectie 4.1.1 stond een XSLT-programma opgesteld in de abstracte syntax. We zullen nu een volledige run van dat programma geven op de inputboom gegeven in figuur 4.7. De verschillende stappen zijn uitgetekend in figuren 4.8, 4.9, 4.10 en 4.11.

We hebben hier een aantal afkortingen gebruikt, die intuïtief wel duidelijk zijn. De context van een knoop staat vermeld tussen hoekige haakjes. In sommige bomen hebben we bepaalde deelbomen vervangen door een driehoekje omdat de bomen anders veel te groot werden. De exacte deelboom kan gevonden worden in de regels van het programma. De context van iedere knoop in zo'n boom staat vermeld onder het driehoekje. Er is wel voor gezorgd dat die deelbomen nooit verder uitgeschreven moeten worden, dus voor het begrijpen van de werking is het weglaten van die deelboom normaal geen probleem. Merk op dat sommige stappen uit een aantal deelstappen bestaan. Dit zijn de bewerkingen van het mini-rewrite-system.

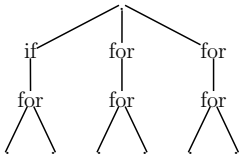
4.2.4 Waarom een mini-rewrite-system?

We splitsten de semantiek van XSLT 1.0 in twee verschillende herschrijfsystemen. Wat is hier de reden voor? Het probleem zit in oneindige berekeningen van templates. We leggen het nut van het mini-rewritesysteem uit aan de hand van een klein voorbeeldje. Beschouw een if-thenstatementknoop waarvan de berekening van de bijbehorende template t een oneindige berekening vereist. De test horende bij de statementknoop is echter steeds false. Wanneer we het mini-rewrite-system niet geïntroduceerd zouden hebben, was het mogelijk dat er steeds verder gewerkt werd in de template t. Door, zodra dat

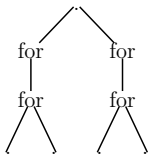
we controlestatementknopen introduceren ze ook meteen wegwerken, vermijden we dergelijke toestanden.



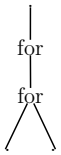
(a) stap 1



(b) stap 2



(c) stap 3

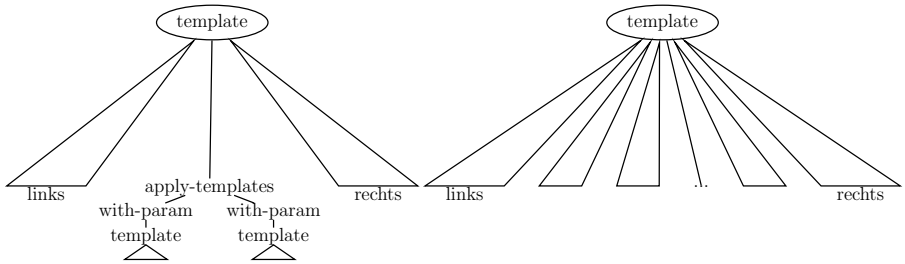


(d)
stap 4

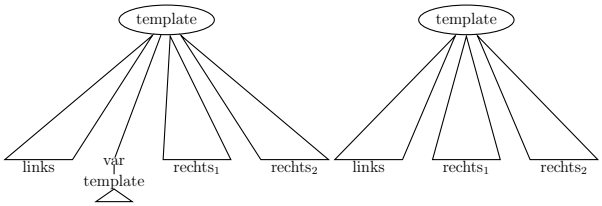
(e)
stap 5

(f) stap 6

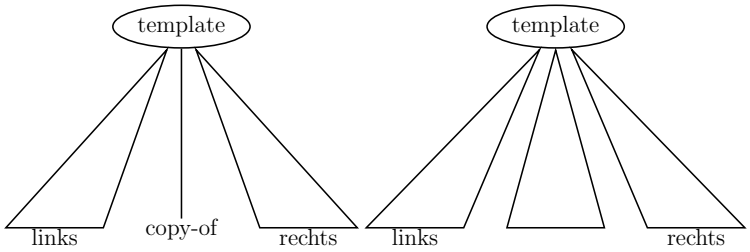
Figuur 4.3: Uitwerking van het mini-rewrite-system



Figuur 4.4: Bomen ϕ_0 en ψ_0 bij de uitwerking van een apply-templates-statementknoop

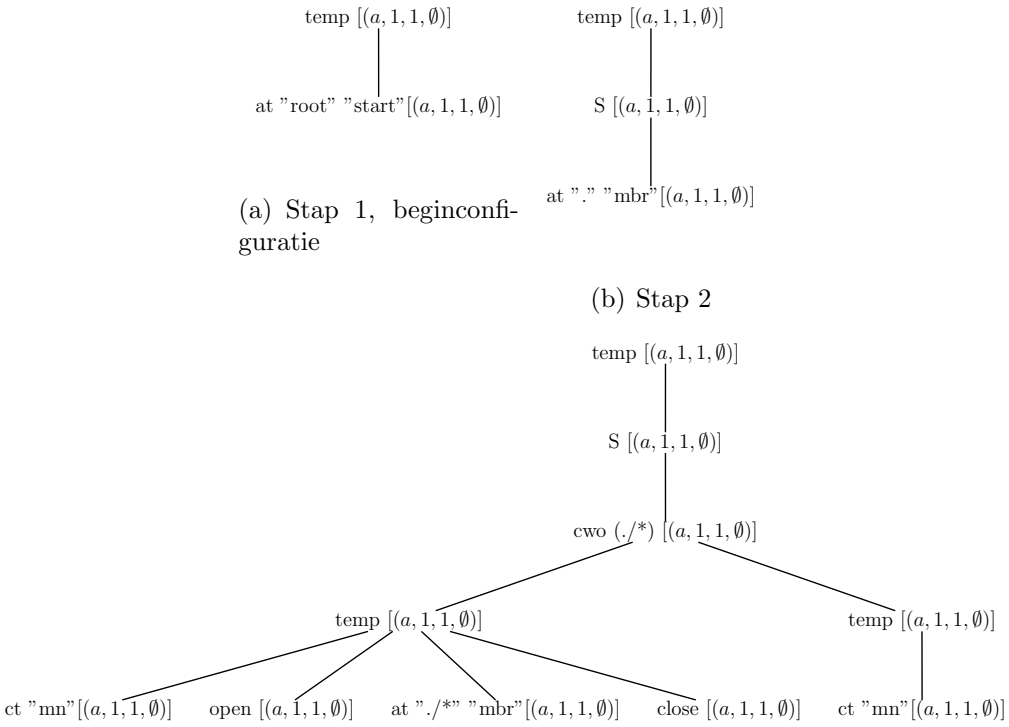


Figuur 4.5: Bomen ϕ_0 en ψ_0 bij de uitwerking van een variabelestatement-knoop



Figuur 4.6: Bomen ϕ_0 en ψ_0 bij de uitwerking van een copy-ofstatementknoop

Figuur 4.7: Inputboom voor uitvoering XSLT programma uit sectie 4.1.1



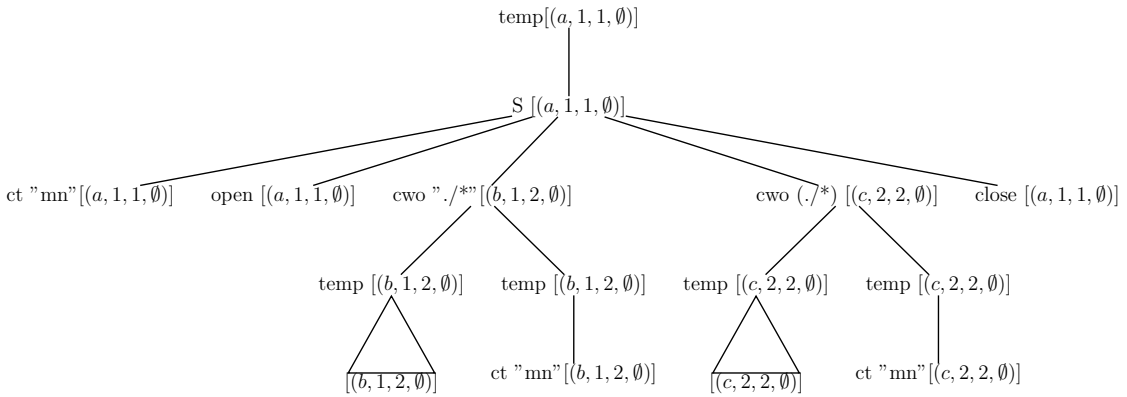
(a) Stap 1, beginconfiguratie

(b) Stap 2

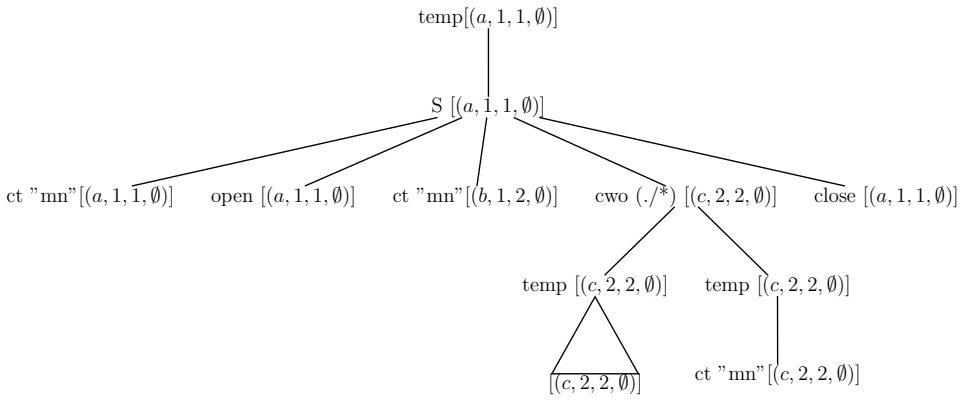
(c) Stap 3.1

(d) Stap 3.2

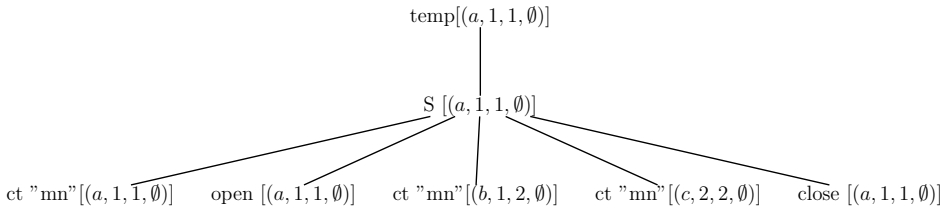
Figuur 4.8: Uitwerking XSLT-programma uit sectie 4.1.1 op de inputboom uit figuur 4.7, deel 1



(a) Stap 4.1

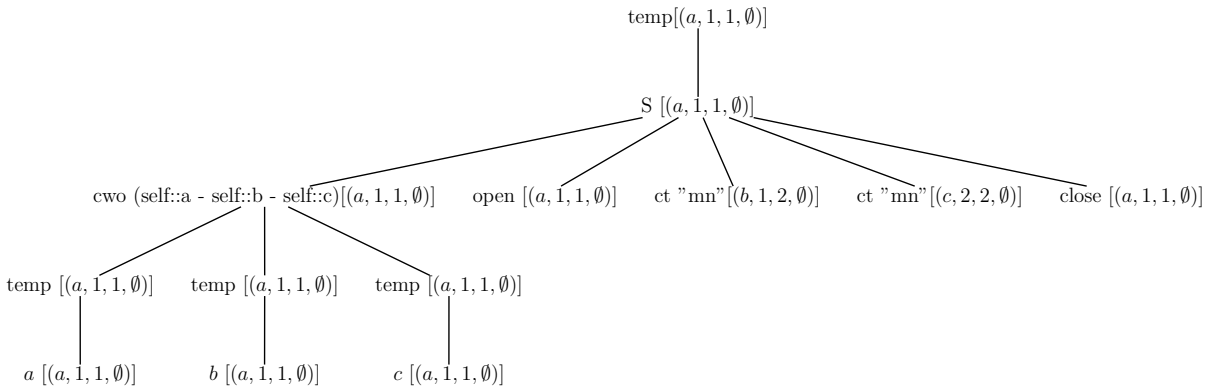


(b) Stap 4.2

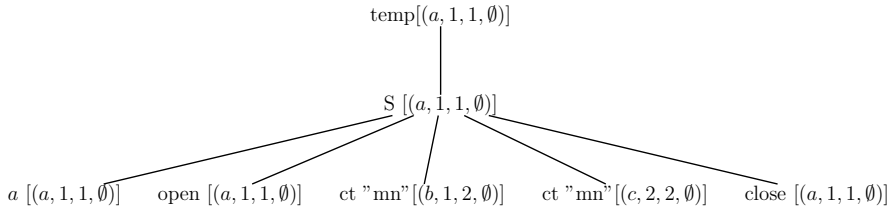


(c) Stap 4.3

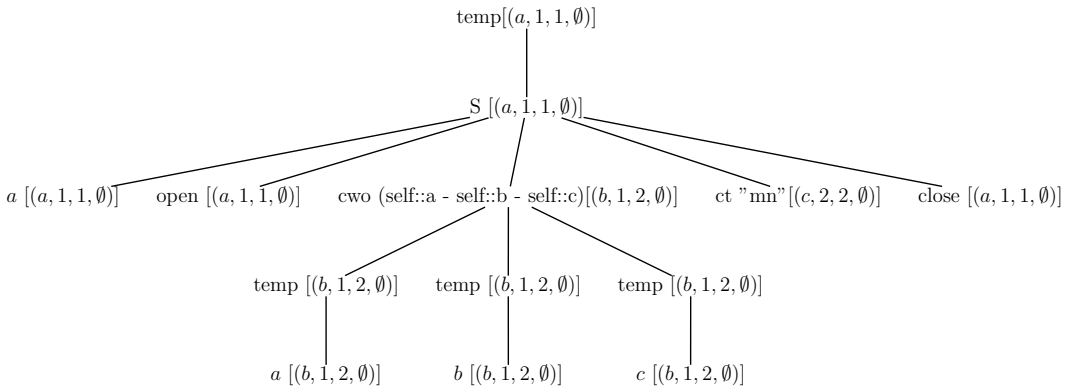
Figuur 4.9: Uitwerking XSLT-programma uit sectie 4.1.1 op de inputboom uit figuur 4.7, deel 2



(a) Stap 5.1

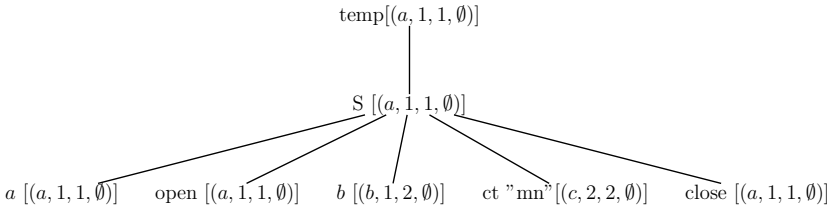


(b) Stap 5.2

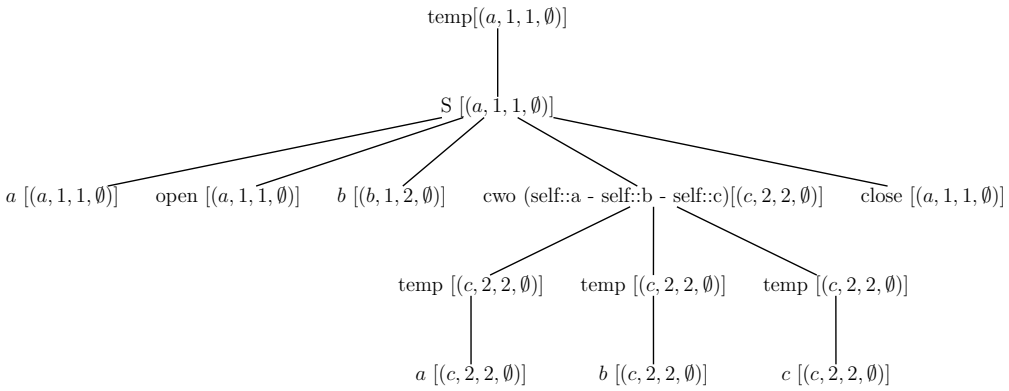


(c) Stap 6.1

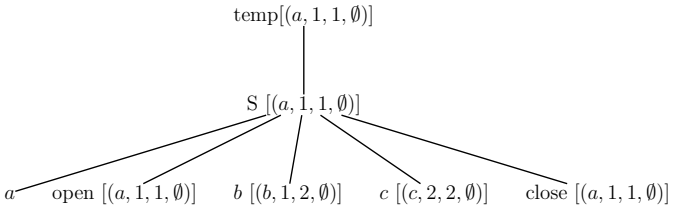
Figuur 4.10: Uitwerking XSLT-programma uit sectie 4.1.1 op de inputboom uit figuur 4.7, deel 3



(a) Stap 6.2



(b) Stap 7.1



(c) Stap 7.2

Figuur 4.11: Uitwerking XSLT-programma uit sectie 4.1.1 op de inputboom uit figuur 4.7, deel 4

Hoofdstuk 5

Terminatie van XSLT-programma's

In dit hoofdstuk trachten we een algoritme te ontwikkelen dat op een gegeven XSLT 1.0 programma en een gegeven inputboom beslist of de transformatie stopt. Aan de hand hiervan proberen we ook een complexiteitsgrens op XSLT 1.0 te leggen. Uit de hiërarchiestelling kunnen we dan afleiden dat bepaalde transformaties niet mogelijk zijn met behulp van XSLT 1.0.

5.1 Inleiding

We hebben essentieel acht statements in onze formele semantiek, namelijk

1. if-then
2. choose-when-otherwise
3. copy-of
4. variabele
5. with-param
6. for-each
7. apply-templates
8. call-template

Hiervan kunnen de eerste zes statementknopen duidelijk nooit de aanleiding geven tot een oneindige lus. Wanneer bijvoorbeeld een if-then tot

```
<?xml version="1.0"?>

<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:template match="/" mode="start">
        <a>
            <xsl:call-template name="functie"/>
        </a>
    </xsl:template>

    <xsl:template name="functie">
        <xsl:call-template name="functie"/>
    </xsl:template>

</xsl:stylesheet>
```

Figuur 5.1: Eerste voorbeeld van oneindigheid

true geëvalueerd wordt en hierdoor een oneindige uitvoering start, ligt de oorzaak bij de template horende bij de if-test. De if-test op zich veroorzaakt de oneindigheid dus niet.

In een programmeertaal, zoals bijvoorbeeld Java, kan een for-eachstatement eenvoudig leiden tot een oneindige lus, door een test te specificeren die steeds tot 'true' geëvalueerd wordt . In XSLT is dit echter niet zo: bij het begin van de for moet je reeds het aantal iteraties opgeven. Dit aantal is zeker eindig omdat er een aantal knopen uit de eindige inputboom worden geselecteerd.

De laatste twee statements, call-template en apply-templates, kunnen wel tot oneindige programma's leiden. Bij een call-template is dat vrij duidelijk. Wanneer binnen de template van een regel met naam *naam* opnieuw de template met naam *naam* wordt opgeroepen, zonder dat de context veranderd is, lopen we oneindig. Figuur 5.1 is hiervan een voorbeeldje. De herhaling valt natuurlijk onmiddellijk op. Wanneer de nieuwe oproep van de template met naam 'functie' verscholen zat in een template die werd opgeroepen met behulp van een apply-templates, is dit echter niet zo vanzelfsprekend.

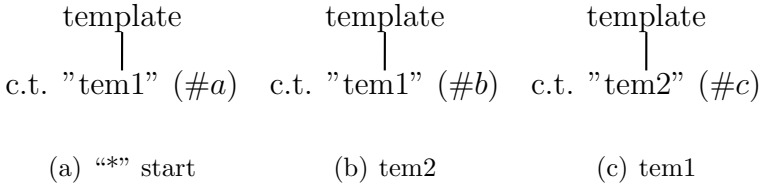
Bij een apply-templatesstatement hebben we eigenlijk hetzelfde probleem: indien binnen de template dezelfde template met dezelfde context wordt op-

geroepen, lopen we oneindig. Dit is echter op het eerste zicht niet zo duidelijk omdat er geen naam meer bij de template hoort. Er moet onder andere gecontroleerd worden of de mode hetzelfde is (wat nog eenvoudig is), maar ook of het pattern horende bij het apply-templatesstatement de huidige knoop selecteert.

5.2 Detectie van oneindigheid

Het idee om oneindigheid te detecteren, is om van iedere knoop bij te houden van welke knoop het een vervanging is en van welke knoop het een kopie is uit de definitie van het programma. We leggen dat uit aan de hand van twee voorbeelden. In de verschillende bomen zijn de knopen gelabeld met een nummer of een letter (bvb. (#4)). Deze nummering dient enkel ter referentie en behoort niet tot de syntax van de stylesheet.

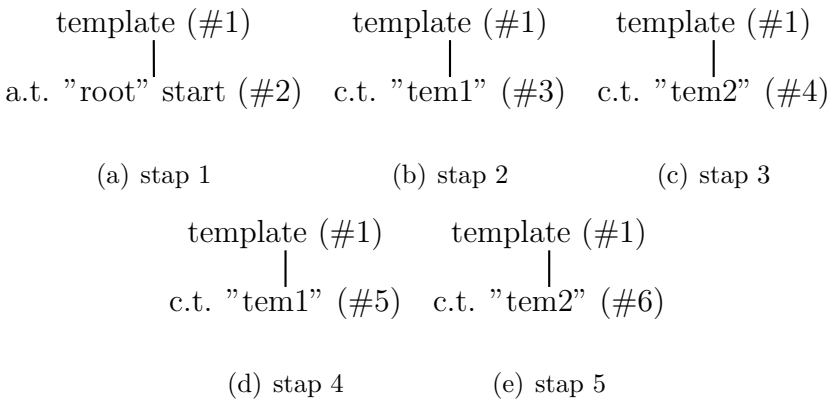
Voorbeeld 5.1. Beschouw het programma gedefinieerd in figuur 5.2 (waarbij c.t. een afkorting is voor call-template). De eerste template heeft als pattern de expressie '*' en als mode 'start'. De tweede template draagt de naam 'tem2', terwijl 'tem1' de naam is van de rechtse template. Een gedeeltelijke uitvoering van dit programma op een willekeurige boom staat beschreven in figuur 5.3.



(a) "*" start (b) tem2 (c) tem1

Figuur 5.2: Programma bestaande uit drie regels

De context van iedere knoop is in dit voorbeeld steeds het tupel bestaande uit de root als contextknoop, de waarde 1 voor de contextpositie en contextsize en een lege variabeletoekenning. We houden in onderstaande tabel de vervangingen en de kopies bij.



(a) stap 1 (b) stap 2 (c) stap 3

(d) stap 4 (e) stap 5

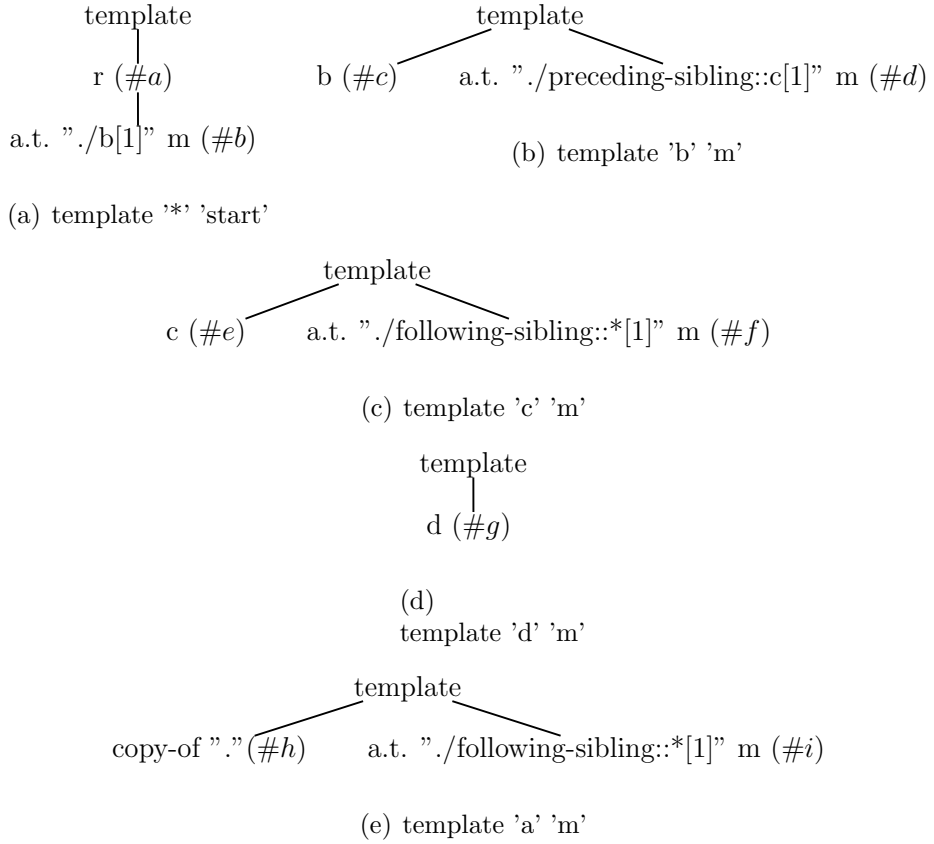
Figuur 5.3: Vijf stappen van de uitvoering van het programma uit figuur 5.2 op willekeurige inputboom.

k	vervangt	kopie van
1	–	–
2	–	–
3	2	a
4	3	c
5	4	b
6	5	c

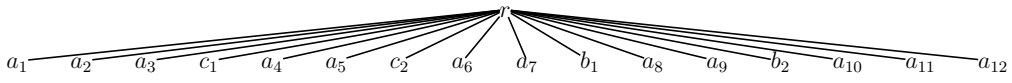
Knoop 6 is een (onrechtstreekse) vervanging van knoop 3. Beide knopen hebben dezelfde context en zijn kopies van dezelfde knoop c uit de definitie van het programma. Deze knoop is een call-template knoop. We mogen besluiten dat ons programma oneindig loopt. ■

Voorbeeld 5.2. Het vorige voorbeeldje was een stylesheet die op gelijk welke input oneindig zou lopen. In het volgende voorbeeld is de oneindigheid eerder het gevolg van een 'slechte' inputboom. De stylesheet begint alle knopen te kopiëren vanaf de eerste c die voor de eerste b voorkomt. Dit wordt gedaan totdat de eerste d-knoop bereikt wordt. Op de boom $r(a_1, a_2, a_3, c_1, a_4, a_5, c_2, a_6, a_7, d_1, a_8, b_1, a_9, a_{10}, b_2, a_{11}, a_{12}, a_{13})$ (waarbij de indexen enkel ter referentie vermeld staan) zal de stylesheet de boom $r(b_1, c_2, a_6, a_7, d_1)$ teruggeven. Het programma loopt dus al zeker niet altijd oneindig. Het probleem duikt echter op wanneer er tussen de geselecteerde c (in het vorige geval c_2) en de eerste b-knoop geen knoop d is. Bij de b-knoop wordt er dan opnieuw gesprongen naar dezelfde c-knoop (c_2), wat aangeeft dat het programma niet stopt. De vijf rules die samen dit programma vormen, zijn getekend in figuur 5.4. De verschillende patternen horende bij deze

regels zijn respectievelijk '*', 'b', 'c', 'd' en 'a'. De eerste regel heeft mode 'start', terwijl de andere allemaal mode 'm' hebben.



Figuur 5.4: Tweede voorbeeld van oneindigheid



Figuur 5.5: Inputboom voor het programma in figuur 5.4

De eerste zeven stappen van de uitvoering op de boom gegeven in figuur 5.5 worden gegeven in figuur 5.6.

We tonen nu aan dat eindigheid van deze stylesheet ook ontdekt kan worden met dezelfde tactiek van vervangingen bijhouden.

De context van een knoop k bestaat steeds uit de contextpositie 1, contextsize 1 en de lege variabeletoekenning. De contextknoop van iedere knoop k is gegeven in de volgende tabel:

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14
contextknoop	r	r	r	r	b_1	b_1	c_2	c_2	a_6	a_6	a_7	a_7	b_1	b_1

k	vervangt	kopie van
1	–	–
2	–	–
3	2	a
4	2	b
5	4	c
6	4	d
7	6	e
8	6	f
9	8	h
10	8	i
11	10	h
12	10	i
13	12	c
14	12	d

Knopen 6 en 14 zijn kopies van dezelfde knoop en hebben dezelfde context. Bovendien is knoop 14 een (onrechtstreekse) vervanging van knoop 6. Het zijn beide apply-templatesstatementknopen, waardoor we mogen besluiten dat het programma oneindig zal lopen. ■

In de vorige voorbeelden kwamen geen variabelen aan bod. Deze zitten echter ook in de context. Indien we hierover geen goede afspraken maken, is de tactiek niet altijd in staat oneindigheid te ontdekken. In het eerste voorbeeld (figuur 5.1) zouden we telkens een parameter kunnen meegeven aan de template met naam 'functie'. Bij iedere aanroep verhogen we de waarde van deze parameter met 1. Alzo komen we nooit dezelfde context tegen. In de semantiek hadden we echter afgesproken dat de maximale waarde van een variabele van type teller nooit groter kon zijn dan het aantal knopen in de inputboom. We spreken dus af dat de bewerkingen op parameters van het type teller modulair worden uitgevoerd (modulo n, met n het aantal knopen in de inputboom). We zouden als parameter ook een variabele van het type result-tree-fragment kunnen meegeven. Iedere nieuwe oproep breidt deze uit met minstens één knoop. Omdat variabelen van dit type echter geen invloed kunnen hebben op de uitvoering, mogen we deze buiten beschouwing laten.

5.3 Besluit (algoritme)

Volgende opsomming is een formelere beschrijving van het algoritme dat checkt of een transformatie stopt op een inputboom:

1. Geef in de beschrijving van de transformatie (de verschillende rules) iedere knoop een unieke letter.

2. Tijdens de uitvoering geef je iedere knoop in de template (behalve de knopen met label 'template') een uniek nummer. Nummers van verwijderde knopen mogen niet herhaald worden.

3. Tijdens de uitvoering van het programma worden er voor iedere knoop, die allemaal uniek genummerd zijn door puntje 2, twee parameters bijgehouden:

   - vervangingsknoop: één knoopnummer (puntje 2). Dit is de knoop die de huidige knoop voorafging. Iedere knoop, buiten de startknopen, in de huidige template van de uitvoering, is geïntroduceerd na het verwijderen van een knoop (zoals in de semantiek overal uitgelegd werd). Het nummer n van deze verwijderde knoop wordt dus bewaard.
   - kopie: één knoopletter uit de definitie van het programma (puntje 1) waarvan de huidige knoop een kopie is.

4. Wanneer je een 'apply-templates' of een 'call-template' knoop k tegenkomt, creëer dan zijn *vervangingsverzameling*. Dit is een verzameling knoopnummers V, die als volgt opgebouwd wordt: initialiseer V met de vervangingsparameter van k. Vervolgens wordt voor iedere knoop $v_i \in V$ de vervangingsparameter van v_i toegevoegd, totdat V niet meer verandert. Controleer of V een knoop n bevat, waarvan de parameterkopie gelijk is aan de kopie-parameter van k. Is dit het geval en de context van knoop k is gelijk aan de context van knoop n, waarbij we de variabelen van het type result-tree-fragment negeren, dan kan er besloten worden dat de stylesheet oneindig zal lopen op de huidige input.

5.4 Vervangingsboom

We zullen het algoritme hier op een andere manier beschrijven, om zo de correctheid aan te tonen. Het algoritme voor terminatie te onderzoeken stelt

een boom op, de vervangingsboom. Het is belangrijk in het achterhoofd te houden dat deze boom verschillend is van de resultaatboom. Deze laatste is die boom waarop we de step-relatie steeds toepassen en die op het einde van een run van een XSLT-programma het resultaat bevat. In de vervangingsboom komen dezelfde knopen voor als in de resultaatboom, maar ze zullen nooit verwijderd worden. Het idee is dat wanneer we knopen introduceren in de resultaatboom, deze ook ergens toevoegen in de vervangingsboom en wel zo dat de nieuwe knopen rechtstreekse afstammelingen zijn van een knoop die verwijderd werd uit de resultaatboom. We geven nu de regels hoe we deze vervangingsboom opbouwen.

- Initieel bevat de boom als root een kopie van de apply-templatesstatementknoop die gecreëerd werd om de uitvoering te starten. De context is niet gedefinieerd.

- Bij het verwijderen van een variabelestatementknoop of een copy-of-statementknoop uit de resultaatboom, verandert er niets in de vervangingsboom.

- Bij het herschrijven van een call-templatestatementknoop in de resultaatboom, zoek je de kopie van die call-template in de vervangingsboom, noem deze knoop k. k krijgt nu dezelfde context als in de resultaatboom. Voor het overige maak je kopies van alle knopen die nieuw toegevoegd zijn in de resultaatboom. Al deze kopies worden toegevoegd in de vervangingsboom als rechtstreekse kinderen van k en krijgen geen context. Voor iedere kopie houden we ook bij van welke knoop uit de definitie van het programma of de inputboom het een kopie is.

- Bij het herschrijven van een apply-templatesstatementknoop in de resultaatboom, doen we een analoge bewerking als bij een call-templatestatementknoop.

Wanneer het programma voor de start van de omzetting een reeks globale variabelen berekent, maken we voor iedere variabele een aparte vervangingsboom. Initieel is deze boom een call-templatestatementknoop die de template oproept die speciaal gemaakt werd voor de berekening. Voor het overige verandert er niets aan de werking. Telkens wanneer we een volgende variabele berekenen, alsook bij de start van de eigenlijke omzetting, beginnen we terug met een lege vervangingsboom.

opmerking 5.1. De vervangingsboom van een XSLT-programma Π op een inputboom t is een eindig-vertakte boom (zie definitie 4.14). Iedere knoop n heeft namelijk als kinderen alle knopen die ingevoegd werden in de resultaatboom bij het verwijderen van n. Dit zijn er een eindig aantal.

lemma 5.1. Zij Π een XSLT-programma en t een willekeurige inputboom. Zij v de vervangingsboom op een bepaald moment van de uitvoering van Π op t. Indien er op een pad vanuit de root van v een call-template- of een apply-templatesstatementknoop herhaald wordt, dan loopt de uitwerking van Π op t oneindig. Met een knoop herhalen wordt hier bedoeld dat het label, de context en de verwijziging naar de originele knoop, identiek zijn, waarbij de variabelen van het type result-tree-fragment genegeerd worden.

Bewijs. Omdat we weten dat XSLT 1.0 unequivocal (deterministisch) is, weten we dat iedere combinatie van een knoop en een context op dezelfde manier herschreven wordt. Waneer we dan bij de herschrijving van een knoop en zijn context opnieuw dezelfde combinatie van die knoop en context tegenkomen, gaan we deze, wegens het determinisme, op dezelfde manier herschrijven. Uiteraard kunnen we dit blijven herhalen en lopen we dus bijgevolg oneindig.

Herhalen is echter gedefinieerd zodat er geen rekening gehouden wordt met result-tree-fragments en dus ook niet met de volledige context. Toch kunnen we dezelfde redenering als hierboven gebruiken, omdat variabelen van het type result-tree-fragment op geen enkele manier invloed kunnen uitoefenen op de uitwerking van een knoop. □

lemma 5.2. Zij Π een XSLT-programma en t een willekeurige inputboom. Zij v de vervangingsboom op een bepaald moment van de uitvoering van Π op t. Wanneer de uitvoering van Π op t oneindig loopt, dan wordt er op een pad van v een knoop herhaald.

Bewijs. Een eerste opmerking die we maken is dat het aantal verschillende labels dat kan voorkomen in de vervangingsboom (en dus ook in de resultaatboom) eindig is. Een label bestaat uit een kopie van een knoop uit Π of t en een context. Het aantal knopen in Π en t is eindig. Ook het aantal verschillende contexten is eindig. Een context bestaat uit vier delen, een contextnode, een contextposition, een contextsize en een variabeletoekenning. Voor de eerste drie onderdelen zijn er hooguit n verschillende waardes, waarbij n het aantal knopen is van t. Voor het overige hebben we nog de variabelen van het type teller. Omdat we in de semantiek zagen dat deze hooguit een waarde n kunnen aannemen, zijn er ook maar een eindig aantal mogelijke variabeletoekenningen voor tellers. De node-set-variabelen tenslotte zijn altijd een subset van de knopen uit t. Bijgevolg zijn er ook maar 2^n mogelijke toekenningen voor een node-set. De variabelen van het type result-tree-fragment mogen we buiten beschouwing laten, omdat deze toch genegeerd worden bij het zoeken naar een herhaling in een vervangingsboom.

We weten dat Π oneindig loopt op t. Hierbij kunnen we dan wel enkele belangrijke opmerkingen maken. We weten namelijk dat het onmogelijk is

dat we oneindig keer een variabele- of een copy-ofstatement herschrijven. Dit is eenvoudig in te zien. Stel dat je op een bepaald moment in de vervangingsboom enkel nog variabele- en copy-ofstatementknopen wegwerkt. Bij iedere uitwerking verdwijnt de knoop uit de resultaatboom en worden er geen nieuwe statementknopen ingevoerd en bijgevolg ook geen nieuwe variabele- of copy-of-statementknopen. Na een tijd moet je dus noodzakelijkerwijs stoppen met deze uit te werken, omdat er simpelweg geen meer zijn. Dezelfde redenering kunnen we maken voor apply-templates- en call-templatestatementknopen die een lege template oproepen, waarbij we met 'leeg' bedoelen dat de template geen statementknopen bevat. Ook dit kan je maar een eindig aantal keer doen, omdat er dan nooit nieuwe statementknopen geïntroduceerd worden.

We kunnen dus besluiten dat, omdat we oneindig lopen, we ook oneindig aantal keer apply-templates-of call-templatestatementknopen wegwerken, waarvan de template niet leeg is. Deze uitwerking zorgt steeds voor een groei van de vervangingsboom. We krijgen dus een oneindige vervangingsboom. Bovendien weten we ook dat een vervangingsboom eindig-vertakt is. We kunnen dus König's lemma (lemma 4.2) toepassen: er moet een oneindig pad zijn in de vervangingsboom. Op dit oneindig pad kunnen er echter slechts een eindig aantal mogelijke labels voorkomen (zoals we aantoonden in het begin van dit bewijs). Er moet dus een herhaling van knoop voorkomen, waarmee het bewijs afgelopen is. □

Wanneer we nu lemma's 5.1 en 5.2 combineren, krijgen we volgende stelling:

stelling 5.1. Zij Π een XSLT-programma en t een willekeurige inputboom. De uitvoering van Π op t loopt oneindig als en slechts als er tijdens die uitvoering op een pad in de vervangingsboom v een knoop herhaald wordt.

opmerking 5.2. In de vorige stelling hielden we een verwijzing bij naar de kopie van de originele knoop uit de definitie van het programma. De eis dat deze waarde hetzelfde moet zijn bij het onderzoek naar een herhaling, is strikt genomen overbodig. Het zal echter wel de berekeningen in verband met de complexiteit van dit algoritme sterk vereenvoudigen. In de volgende voorbeelden, waar de verwijzing dus geen rol speelt, is deze dan ook weggelaten.

Voorbeeld 5.3. Hier hernemen we voorbeeld 5.1, maar we testen op oneindigheid met behulp van de vervangingsboom. In figuur 5.7 staan de verschillende stappen getekend. Omdat iedere knoop in de boom als context $(root, 1, 1, \emptyset)$ heeft, is deze niet steeds uitdrukkelijk vermeld. In de laatste

stap dragen de omcirkelde knopen hetzelfde label en is de context gelijk. We mogen dus besluiten dat dit programma oneindig zal lopen. ■

Voorbeeld 5.4. Het tweede voorbeeld van dit hoofdstuk, voorbeeld 5.2, wordt opnieuw uitgewerkt. In figuren 5.8 en 5.9 staat de uitwerking van de vervangingsbooom getekend. De contextsize en contextpositie is steeds 1. De variabeletoekenning is steeds leeg omdat er geen variabelen zijn. Daarom vermelden we bij iedere knoop, waarvan de context gedefinieerd is, enkel de contextknoop (tussen haakjes).

Hier zien we opnieuw dat de omcirkelde knopen zowel hetzelfde label als dezelfde context hebben. Het programma zal dus oneindig lopen. ■

5.5 Complexiteit

We willen berekenen hoe groot de vervangingsboom v van een programma $\Pi = (V, \mathit{type}, R, <, <_g, \mathit{def})$ op een inputboom t kan worden. Zij n het aantal knopen in t en m het aantal knopen in alle regels van Π samen. Zij p het aantal variabelen van het type node-set en q het aantal variabelen van het type teller in V. Om het aantal mogelijke knopen te berekenen, moeten we weten hoeveel verschillende contexten er bestaan. Het aantal mogelijke contextknopen dat kan voorkomen is gelijk aan n. Een contextsize is de grootte van een niet-lege verzameling knopen uit t, dus er bestaan ook hooguit n verschillende contextsizes. Omdat de contextpositie altijd een getal tussen 1 en de contextsize is, zijn er ook hooguit n contextposities. In de semantiek van XSLT 1.0 spraken we af dat een teller maximale waarde n heeft. Voor iedere teller bestaan er dus $n + 1$ verschillende toewijzigingen. Een variabele van het type node-set is een deelverzameling van de n knopen, bijgevolg bestaan er 2^n mogelijke waardes voor een node-set-variabele. Het aantal mogelijke variabeletoekenningen is bijgevolg gelijk aan[1] $(2^n)^p \times (n+1)^q$. Wanneer we al deze resultaten vermenigvuldigen, krijgen we het aantal mogelijke contexten (waarbij we result-tree-fragments negeren): $n \times n \times n \times ((2^n)^p \times (n+1)^q) = n^3 \times ((2^n)^p \times (n+1)^q)$. Op een pad vanuit de root in de vervangingsboom mocht geen herhaling voorkomen van een knoop (anders lopen we oneindig). We berekenen nu een bovengrens op de lengte van een pad vanuit de root. We kunnen alle knopen van het programma onder elkaar zetten. Wanneer al deze knopen dezelfde context hebben, komt er geen herhaling voor (waarbij we *wel* rekening houden met de kopieparameter). We kunnen deze sequentie herhalen met steeds een andere context,

[1] Algemeen: het aantal mogelijke keuzes van b elementen uit een verzameling van grootte a, waarbij de volgorde van belang is en waarbij herhalingen mogen voorkomen, is gelijk aan a^b.

zonder ooit een herhaling te introduceren. Omdat we het aantal contexten kunnen berekenen, $n^3 \times ((2^n)^p \times (n+1)^q)$, en er m knopen zijn, is een pad vanuit de root hooguit $n^3 \times ((2^n)^p \times (n+1)^q) \times (m)$ lang. Noem dit getal r.

Wat is nu de maximale grootte van een volledige vervangingsboom wanneer het programma niet oneindig loopt? We weten dat de boom een maximale diepte r heeft, maar wat is het maximaal aantal kinderen van een knoop? Bij de herschrijving van een variabelestatement zijn er geen kinderen in de vervangingsboom, net zoals bij een copy-ofstatementknoop. Het is hier erg belangrijk dat we bij een copy-ofstatementknoop de nieuwe knopen niet toevoegen in de vervangingsboom. Het zou namelijk kunnen dat een copy-ofstatementknoop een result-tree-fragment invoegt en dan kunnen we geen grens leggen op het maximaal aantal nieuwe knopen. De knopen die een copy-ofstatementknoop introduceert, of dit nu knopen uit de inputboom zijn of uit een result-tree-fragment, zijn zeker geen statementknopen. Bijgevolg zullen deze knopen geen oneindigheid kunnen veroorzaken, zodat het geen erg is deze weg te laten uit de vervangingsboom. Wanneer we echter een call-template- of een apply-templatesstamentknoop uitwerken, worden er wel kinderen geïntroduceerd bij de overeenkomstige knoop in de vervangingsboom. Deze kinderen komen allemaal uit de templates van de regels. Op het eerste zicht kunnen er dit dus hooguit m zijn. Er moet echter ook rekening gehouden worden met for-eachstatementknopen (deze horen strikt genomen niet tot het subtree-replacement-system (zie sectie 4.2.2), maar kunnen er wel een invloed op hebben) en apply-templatesstatementknopen. Deze operaties kunnen elk een variabel aantal keer bepaalde templates toevoegen. Dit aantal is echter ook beperkt tot n, doordat n het maximaal aantal iteraties is van een for-eachstatement en n ook het maximaal aantal knopen is wat een expressie van een apply-templatesstatementknoop kan selecteren. We kunnen stellen dat het maximale aantal nieuwe toegevoegde knopen in eerste instantie dus hooguit $n \times m$ is. In deze nieuwe knopen kunnen echter opnieuw forlussen zitten, zodat deze ook weer hooguit $n \times m$ knopen introduceren. De diepte van een geneste-forlus is echter maximaal m, waardoor in de vervangingsboom iedere knoop dus hooguit $(n \times m)^m$ kinderen heeft. Stel voor de eenvoud van notatie dat $s = (n \times m)^m$.

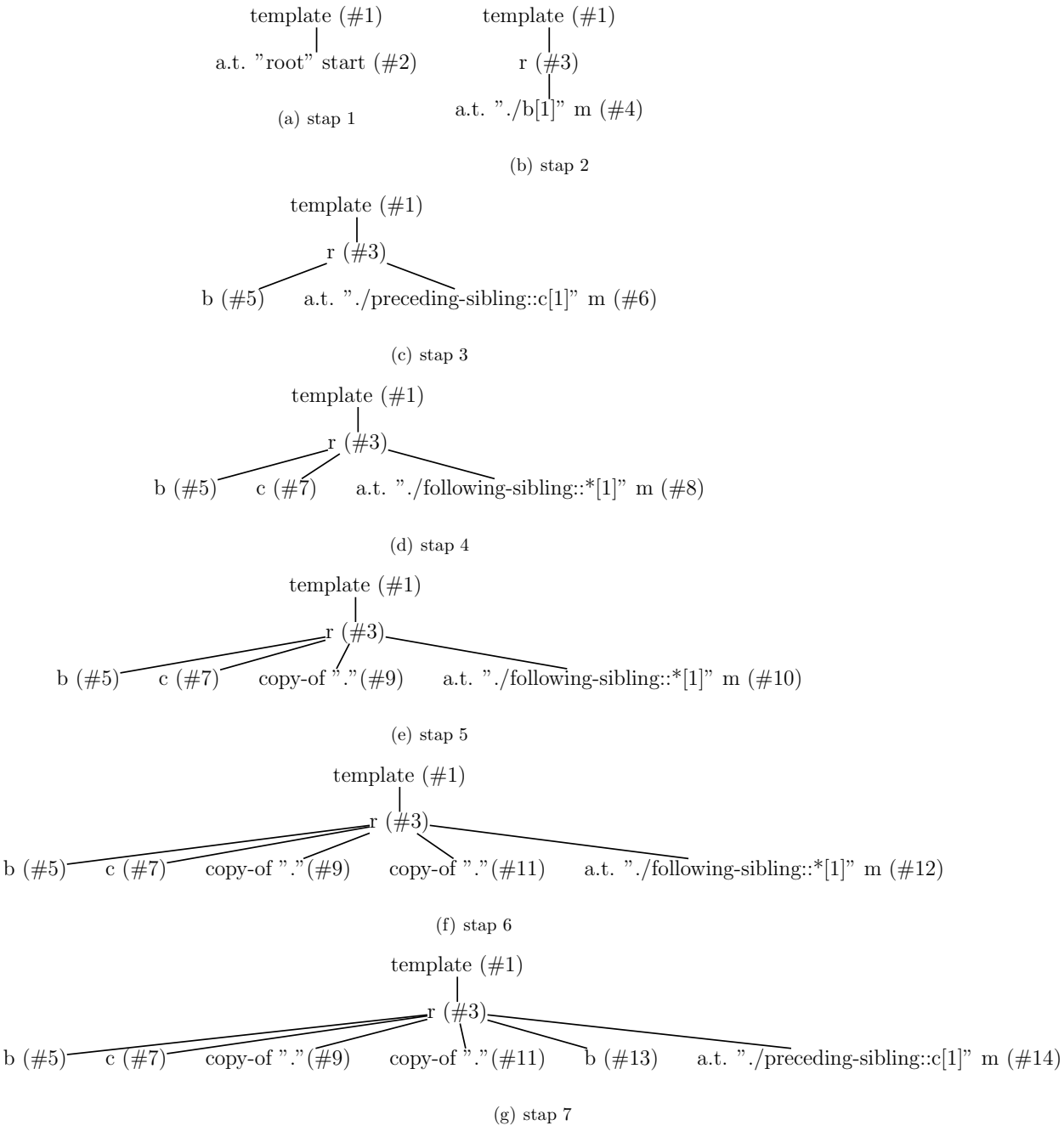
Een vervangingsboom van Π heeft dus hooguit $\sum_{i=1}^{r} (s^{i-1})$ knopen. Als we dit getal uitrekenen, krijgen we $(s)^0 + (s)^1 + \cdots + (s)^{r-2} + (s)^{r-1} = \frac{1-s^r}{1-s}$. Door nu r en s te vervangen door hun formules, krijgen we voor de maximale grootte van een vervangingsboom het getal

$$\frac{1 - ((n \times m)^m)^{(n^3 \times ((2^n)^p \times (n+1)^q) \times (m))}}{1 - ((n \times m)^m)}$$

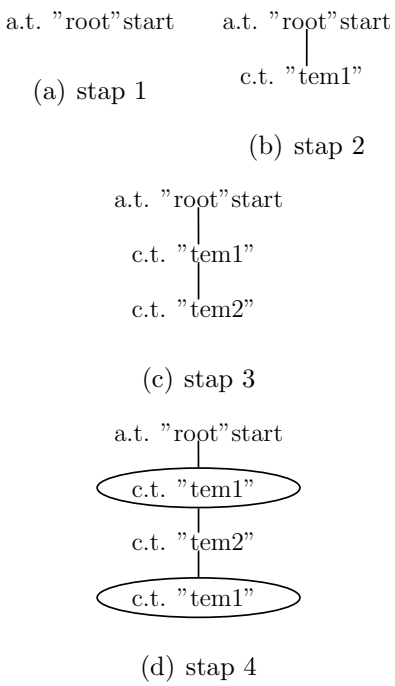
We gaan nu dit getal ook opschrijven in de grote-O - notatie. Merk op dat

voor een gegeven programma Π de variabelen m, p en q allemaal constanten zijn. r kunnen we schrijven als $O(2^n)$ en s als $n^{O(1)}$. De volledige formule wordt dan $n^{2^{O(n)}} = 2^{\log(n)2^{O(n)}} = 2^{2^{O(n)}}$.

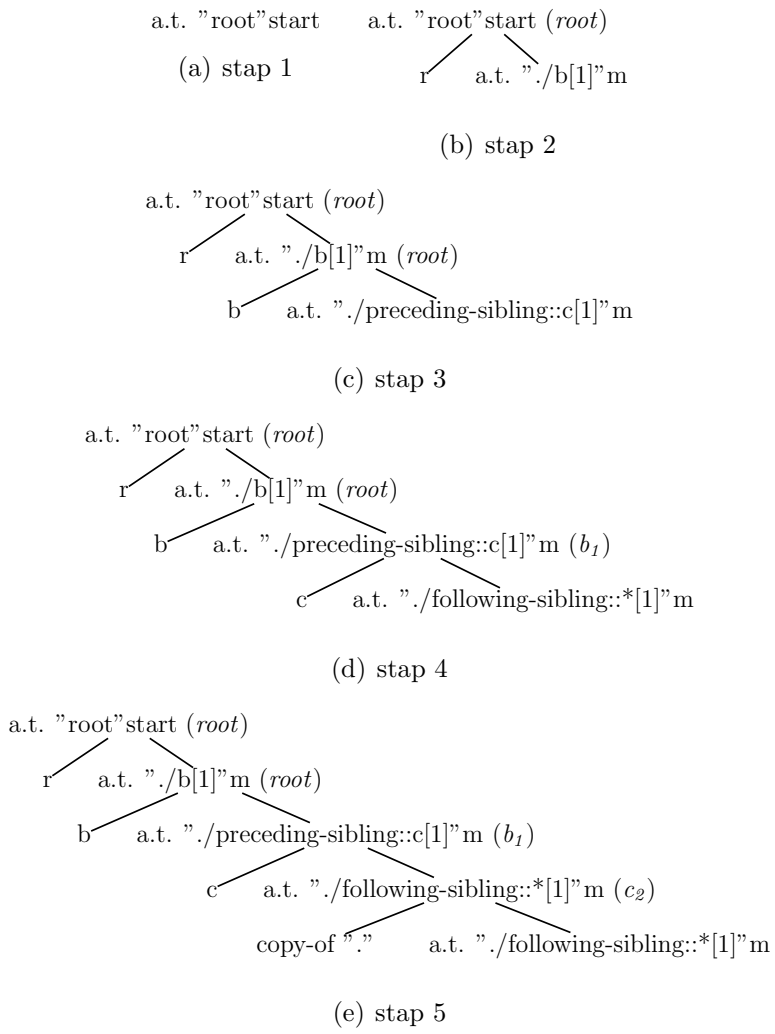
Voorbeeld 5.5. In voorbeeld 4.1.1 stelden we een XSLT-programma op in de abstracte syntax. Het aantal knopen van dit programma is 17. In voorbeeld 4.2.3 simuleerden we een run van dat programma op een boom met 3 knopen. Met de berekeningen van hierboven zien we dat de vervangingsboom van die uitwerking nooit groter kan worden dan (afgerond) $0.3219357957 \times 10^{75499}$ knopen. ■



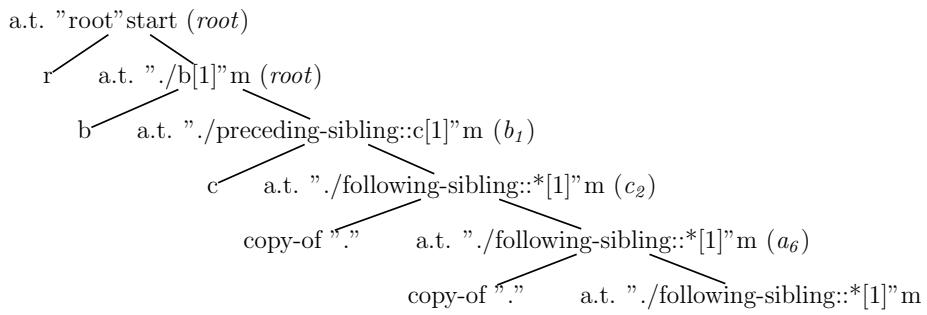
Figuur 5.6: De eerste zeven stappen van het programma gegeven in figuur 5.4 op de inputboom uit figuur 5.5



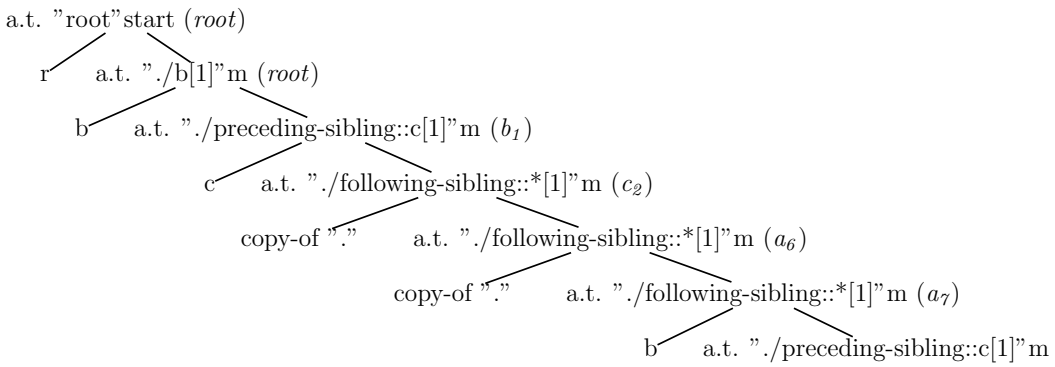
Figuur 5.7: Uitwerking vervangingsboom uit voorbeeld 5.1



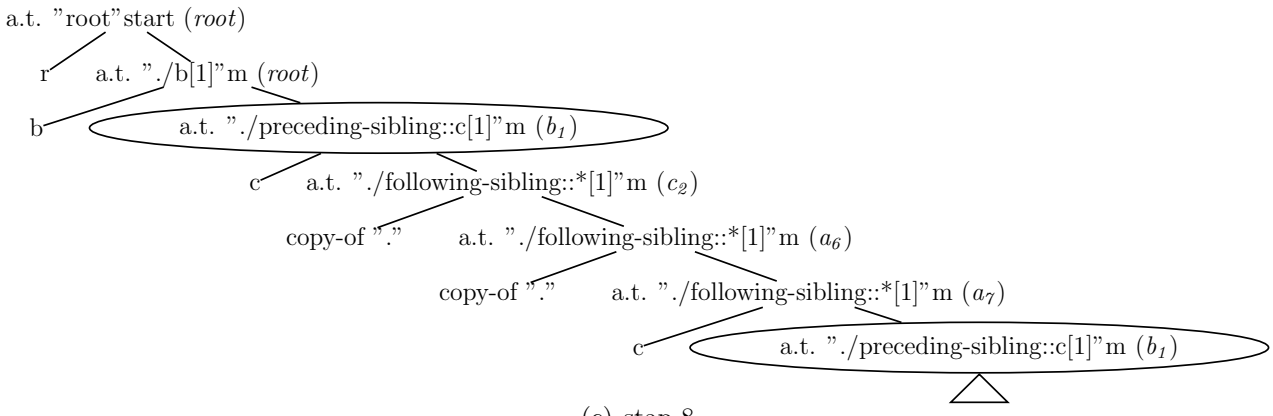
Figuur 5.8: Uitwerking vervangingsboom uit voorbeeld 5.2, deel 1



(a) stap 6



(b) stap 7



(c) stap 8

Figuur 5.9: Uitwerking vervangingsboom uit voorbeeld 5.2, deel 2

Hoofdstuk 6

XSLT 1.0: resultaten en besluit

Wat hebben we nu bereikt met voorgaande hoofdstukken? Een eerste belangrijk resultaat is dat XSLT 1.0 confluent is. Dit maakt het voor XSLT-processoren makkelijk om zich te houden aan de recommendation van XSLT 1.0 [Cla99]. Hierin staat namelijk vermeld dat een XSLT-processor de stylesheet in eender welke volgorde mag uitvoeren, zolang het eindresultaat maar hetzelfde is als een uitvoering die zich aan de volgorde van de recommendation houdt *(Implementations are free to process the source document in any way that produces the same result as if it were processed using this processing model.)*. In hoofdstuk vier toonden we formeel aan dat iedere willekeurige volgorde dit resultaat zal opleveren.

Een tweede resultaat is dat we een complexiteitsgrens kunnen leggen op XSLT 1.0. Dit formuleren we met behulp van onderstaande stelling:

stelling 6.1. Zij Π een willekeurig XSLT 1.0-programma en beschouw het volgende probleem *DEF*Π:

input:	een willekeurige boom t
output:	$\Pi(t)$ als dit gedefinieerd is, ofwel ∞.

Het probleem *DEF*Π zit vervat in SPACE($2^{O(n)}$).

Bewijs. Zij Π een willekeurig XSLT-programma en zij t een willekeurige boom met n knopen. Stel dat Π zelf m knopen bevat en in totaal p variabelen van het type teller gebruikt en q variabelen van het type node-set. Een eerste manier waarop we *DEF*Π kunnen oplossen, is het starten van de uitvoering om het resultaat van Π op t te achterhalen. Hierbij werken we nooit copy-ofstatementknopen uit. Deze statementknopen kunnen geen oneindigheid veroorzaken, dus op zich is het niet erg deze uitwerking uit te stellen. Zodra tijdens de uitvoering de vervangingsboom de grens

van $\frac{1-((n\times m)^m)^{(n^3\times((2^n)^p\times(n+1)^q)\times(m))}}{1-((n\times m)^m)}$ knopen overschrijdt, weten we dat het programma oneindig zal lopen en zullen we dus '∞' outputten. Doordat we copy-ofstatementknopen niet uitwerken in de resultaatboom, zal die boom ook zeker die grens op het aantal knopen niet overschrijden. Echter hebben we hiermee niet aangetoond dat *DEFΠ* $\in$ SPACE($2^{O(n)}$), maar wel dat *DEFΠ* $\in$ SPACE($2^{2^{O(n)}}$).

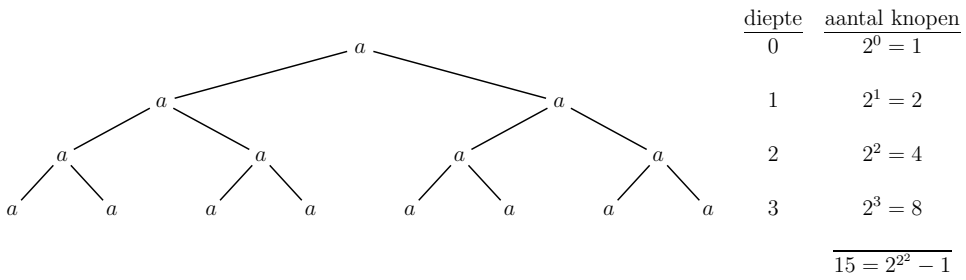
We weten dat XSLT 1.0 confluent is. Iedere mogelijke volgorde van uitvoering levert dus hetzelfde resultaat op. Wanneer we nu steeds één pad uitwerken van de vervangingsboom en checken of dat pad niet meer knopen bevat dan de maximale padlengte $(n^3 \times ((2^n)^p \times (n+1)^q) \times (m))$, kunnen we ook oneindigheid detecteren. Als uitvoeringsvolgorde kunnen we steeds het meest linkse pad van onuitgewerkte statementknopen volgen. Zodra een dergelijke pad volledig uitwerkt is, kunnen we daarvan het resultaat outputten en doorgaan met het volgende pad. We moeten dus telkens hooguit $n^3 \times ((2^n)^p \times (n+1)^q) \times (m)$ knopen bewaren, samen met hun kinderen. Dit aantal kinderen is beperkt tot $(n \times m)^m$. We kunnen dus concluderen dat *DEFΠ* $\in$ SPACE($2^{O(n)}$). □

We hebben dus aangetoond dat voor elk vast XSLT 1.0 programma Π *DEFΠ* in EXPSPACE zit. Het zou nog beter zijn als we kunnen aantonen EXPSPACE de best haalbare complexiteitsklasse is voor *DEFΠ*. Dit kunnen we doen door op zoek te gaan naar een Π_0, zodat *DEFΠ* niet zit in SPACE(f), met $f \in$ o(2^n).

Uit de hiërarchiestelling van plaatscomplexiteit weten we dat er een Turing machine M bestaat zodat $L(M) \in$ SPACE(2^n) en $L(M) \notin$ SPACE(f) met $f = o(2^n)$. Als we nu in staat zijn M te simuleren op een inputstring s met behulp van een XSLT 1.0-programma Π, dan weten we ook dat *DEFΠ* $\notin$ SPACE(f). De output van *DEFΠ* vertelt of M de inputstring s aanvaardt in space 2^n of niet.

Stel dus dat we voor iedere Turing machine M een XSLT 1.0-programma kunnen construeren die M simuleert binnen space 2^n. In het bijzonder bestaat er dan voor iedere $f =$ o(2^n) een XSLT 1.0-programma zodat *DEFΠ* $\notin$ SPACE(f).

Het is echter niet meteen duidelijk hoe een Turing machine in space 2^n met XSLT 1.0 gesimuleerd kan worden. Een andere manier waarmee we kunnen aantonen dat onze complexiteitsgrens de best mogelijke is, is het uitdokteren van een nieuw berekeningsmodel T, zoals eindige automaten, pushdownautomaten, ... Eerst moet aangetoond worden dat iedere uitvoering van een dergelijk berekeningsmodel gesimuleerd kan worden met XSLT 1.0. In de andere richting moet er aangetoond worden dat voor iedere willekeurige stylesheet Π een instantie I van het berekeningsmodel gecreëerd kan worden



Figuur 6.1: Binaire boom met diepte 2^{2^n}

zodat de uitwerking van I op een boom t hetzelfde resultaat geeft als de uitwerking van Π op t. Vervolgens kan er onderzoek verricht worden naar de complexiteit en kracht van T.

Voorbeeld 6.1. In dit voorbeeld werken we een XSLT-programma uit dat op een inputboom met n knopen een boom oplevert die $2^{2^n} - 1$ knopen bevat. Het idee is om een binaire boom te maken met diepte 2^n. Van zo'n boom weten we dat het aantal knopen gelijk is aan $2^{2^n} - 1$, zoals in figuur 6.1 geïllustreerd wordt (voor $n = 2$).

Hoe maken we nu de boom in XSLT? Een eerste manier zou zijn om een variabele van het type teller bij te houden die aangeeft hoe diep we zitten in de boom. Deze waarde geven we telkens mee aan een functie die de boom niveau per niveau opstelt. Het probleem hierbij is echter dat we in de semantiek hadden afgesproken dat de maximale waarde van een teller gelijk is aan n. In deze aanpak zou de teller lopen tot 2^n, wat deze tactiek onmogelijk maakt.

Het idee is echter node-set-variabelen te gebruiken als diepteteller. We lopen alle mogelijke node-sets af van de inputboom. Omdat de inputboom uit n knopen bestaat, zijn er 2^n verschillende node-sets. We moeten dan nog wel afspreken hoe we naar de 'volgende' node-set gaan. Hiervoor gebruiken we de lexicale volgorde. Voor een inputboom met drie knopen gebruiken we onderstaande volgorde van node-sets:

$$\emptyset \to \{1\} \to \{1,2\} \to \{1,2,3\} \to \{1,3\} \to \{2\} \to \{2,3\} \to \{3\}$$

In pseudo-code is de berekening van de node-set volgende op een gegeven node-set gegeven in algoritme 1. Om dit stukje code echter in XSLT te vertalen, moeten we de testen aanpassen. Een test wordt namelijk als true gezien indien er een niet-lege node-set wordt geselecteerd. Het volgende probleem is het creëren van de node-sets die teruggegeven dienen te worden. We willen de regels 8 tot 13 samenvoegen in één XPath-expressie. Noem

Algoritme 1 Het berekenen van de volgende node-set uit de huidige node-set

Require: node-set parameter: c

1: **if** $c == \emptyset$ **then**
2: return $\{1\}$
3: **end if**
4: **if** $c == \{n\}$ **then**
5: return //er is geen volgende node-set meer
6: **end if**
7: $m = \max(c)$
8: **if** $m < n$ **then**
9: return $c \cup \{m + 1\}$
10: **else**
11: $m_2 = \max(c - \{m\})$
12: return $(c - \{m, m_2\}) \cup \{m_2 + 1\}$
13: **end if**

de test $(m < n)$ e_1, de waarde van het return-statement op regel 9 e_2 en tenslotte het laatste return-statement e_3. De volledige XPath-expressie die dan de regels 8-13 simuleert wordt gegeven door:

$$(e_2[e_1])|(e_3[not(e_1)])$$

. e_1 is het eenvoudigst: m is kleiner dan n als en slechts als er nog knopen (in document-order) volgen op m. Deze test kunnen we dan simuleren als volgt:

```
($currentNodeSet[position()=last()]/descendant::* |
 $currentNodeSet[position()=last()]/following::*)
```

Het resultaat zal leeg zijn als er geen knopen meer volgen na de grootste knoop van de huidige node-set.

De XPath-expressie voor e_2 begint met alle knopen uit de huidige node-set te selecteren. Vervolgens worden alle opvolgers van de laatste knoop van deze set geselecteerd. De eerste knoop uit deze opvolgersverzameling is de opvolger van m.

```
($currentNodeSet |
    (($currentNodeSet[position()=last()]/descendant::* |
        $currentNodeSet[position()=last()]/following::*)
                                            [position() = 1]))
```

Tenslotte moeten we e_3 nog opstellen. e_3 is een unie van 2 delen: e_{3_1} en e_{3_2}. We beginnen met e_{3_1}, die alle knopen van de huidige node-set selecteert, behalve de laatste twee.

```
$currentNodeSet[not(position()=last()-1) and not(position()=last())]
```

e_{3_2} zal aan het resultaat van e_{3_1} nog de knoop toevoegen die in document-order volgt op m_2.

```
((($currentNodeSet[not(position() = last())])
          [position()=last()])/following::*) |
(((($currentNodeSet[not(position() = last())])
          [position()=last()])/descendant::*))[position() = 1]
```

Het volledige XSLT-programma wordt gegeven in figuur 6.2. ■

De basis voor deze resultaten was natuurlijk het formeel definiëren van de XSLT-semantiek. Dit formeel model kan ook nog gebruikt worden om andere resultaten mee te bewijzen. Een voorbeeldje hiervan is onderstaande stelling, die zegt dat de for-eachstatementknoop strikt genomen overbodig is.

stelling 6.2. Voor ieder XSLT 1.0-programma Π bestaat er een XSLT 1.0-programma Π' dat geen for-eachstatementknopen bevat en dat op iedere inputboom t hetzelfde resultaat oplevert als $\Pi(t)$.

Bewijs. Zij Π een XSLT 1.0-programma en zij f een for-eachstatementknoop in een regel van Π. We geven nu een manier om de for-eachstatementknoop weg te werken. Dit wegwerken gaat het resultaat van de uitvoering van Π op een inputboom niet veranderen. Als we nu op dezelfde manier ieder andere for-eachstatementknoop uit Π wegwerken, krijgen we een nieuw XSLT 1.0-programma Π' dat geen for-eachstatementknopen bevat, maar waarbij, voor een willekeurige inputboom t, geldt: $\Pi(t) = \Pi'(t)$.

In figuur 6.3(a) staat de beginsituatie gegeven: een for-eachstatementknoop f met als kind de template *template*. De context van f is $c = (n, cp, cs, \alpha)$. Geen enkel van de afstammelingen van f heeft een context (knopen die afstammelingen zijn van een for-eachstatementknoop hebben nooit een context). Zij e de expressie horende bij f en stel $\text{eval}_{\wp}(e, t, c) = \{n_1, n_2, \ldots, n_p\}$. De uitwerking van f geeft dan de situatie gegeven in figuur 6.3(b).

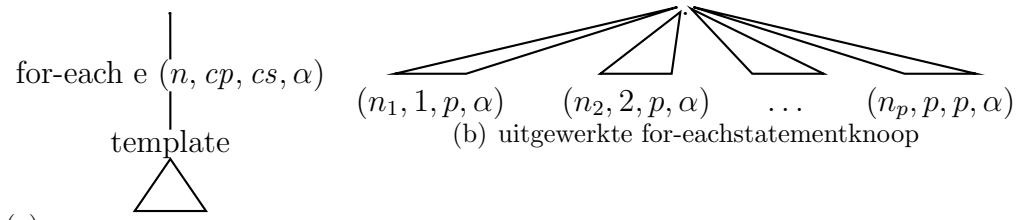
We zullen nu beschrijven hoe we f kunnen omvormen. Eerst voegen we een nieuwe regel toe aan ons programma. Het pattern horende bij deze regel is de expressie uit $\wp$ die alle knopen van de inputboom selecteert. De mode is m, waarbij m een mode-naam is die nog niet eerder voorkwam. De template van deze regel is *template*. Vervolgens verwijderen we in Π de knoop f en al zijn afstammelingen. We vervangen f door de knoop a, waarvan het label 'apply-templates e m' is. Deze situatie is gegeven in figuur 6.4(a). Wanneer we deze knoop a uitwerken met context c, worden natuurlijk

door e dezelfde knopen van t geselecteerd. We gaan dan op zoek naar een regel die we kunnen toepassen. De enigste regel die we vinden is de zonet gecreëerde, omdat deze als enige de mode m heeft. We voegen dus voor iedere knoop die e selecteerde een kopie van de template horende bij deze regel toe. Deze template is hetzelfde als die van f. Voor iedere geselecteerde knoop n_i, $i = 1 \ldots p$, is de context van de overeenkomstige template gelijk aan (n_i, i, p, α). Het resultaat van deze uitwerking is dus hetzelfde als de uitwerking van f.

We hebben dus een manier uitgewerkt die in staat is de werking van een for-eachstatementknoop te simuleren. Hieruit kunnen we dus besluiten dat de for-eachknoop niet primitief is. De for-eachstatementknoop maakt XSLT 1.0 dus niet krachtiger. □

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:template match="*">
            <xsl:call-template name="createBinaryTree">
                <xsl:with-param name="currentNodeSet" select="/.."/>
            </xsl:call-template>
    </xsl:template>

    <xsl:template name="createBinaryTree">
        <xsl:param name="currentNodeSet"/>

        <xsl:variable name="allNodes" select="//*"/>
        <xsl:variable name="lastNode" select="$allNodes[position() = last()]"/>

        <xsl:choose>
            <xsl:when test="$currentNodeSet">
            <!--De currentNodeSet is niet leeg-->
                <xsl:choose>
                    <xsl:when test="$currentNodeSet[not (. = $lastNode)]">
                    <!--De currentNodeSet is niet n-->
                        <xsl:variable name="parNextNodeSet" select="e2[e1] | e3[not(e1)]"/>
                        <a>
                            <xsl:call-template name="createBinaryTree">
                                <xsl:with-param name="currentNodeSet" select="$parNextNodeSet"/>
                            </xsl:call-template>
                        </a>

                        <a>
                            <xsl:call-template name="createBinaryTree">
                                <xsl:with-param name="currentNodeSet" select="$parNextNodeSet"/>
                            </xsl:call-template>
                        </a>
                    </xsl:when>

                    <xsl:otherwise>
                    <!--De currentNodeSet is n-->
                        <a/>
                        <a/>
                    </xsl:otherwise>

                </xsl:choose>
            </xsl:when>

            <xsl:otherwise>
            <!--De currentNodeSet is wel leeg-->
                <a>
                    <xsl:call-template name="createBinaryTree">
                        <xsl:with-param name="currentNodeSet" select="$allNodes[position()=1]"/>
                    </xsl:call-template>
                </a>
            </xsl:otherwise>
        </xsl:choose>
    </xsl:template>
</xsl:stylesheet>
```

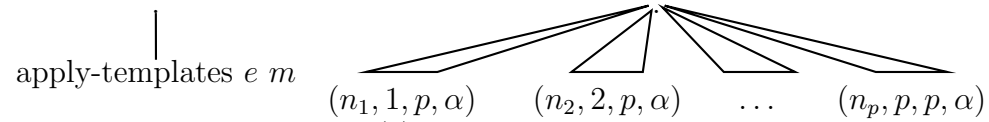
Figuur 6.2: Stylesheet: creëer een boom met $2^{2^n} - 1$ knopen



(a)
een for-eachstatementknoop

(b) uitgewerkte for-eachstatementknoop

Figuur 6.3: Uitwerking van een for-eachstatementknoop f



(a)
apply-templatesstatementknoop

(b) uitgewerkte apply-templatesstatementknoop

Figuur 6.4: Uitwerking van een apply-templatesstatementknoop a

Hoofdstuk 7

XSLT 2.0

Zoals reeds verteld in het inleidende hoofdstuk, werd na de voorstelling van de XSLT 1.0-recommendation gestart met de ontwikkeling van een veel uitgebreidere versie van XSLT. Er werden een heel deel nieuwe features toegevoegd zoals datatypes, meerdere resultaatbomen, functions, deeltijdse bomen (temporary trees), ... Deze 2.0 versie, voorafgegaan door de tussentijdse versie 1.1, is al meer dan vijf jaar in ontwikkeling. Toch zijn de voornaamste toevoegingen en wijzigingen ten opzichte van versie 1.0 reeds uitgewerkt. We zullen in dit hoofdstuk deze veranderingen aanhalen en vervolgens de abstracte syntax en de formele semantiek van XSLT 1.0 (hoofdstuk 4) uitwerken naar een 2.0 versie. We zullen ons echter slechts op één nieuw feature toespitsen, namelijk de temporary trees. Eigenlijk werd dit principe al uitgewerkt in XSLT 1.1, zoals ook beschreven in sectie 1.4.

7.1 Toevoegingen en wijzigingen in XSLT 2.0

XSLT 2.0 werkt samen met XPath 2.0 in plaats van XPath 1.0. De ontwikkelingen van XPath 2.0 en XSLT 2.0 liepen hand in hand. Wanneer in één van beide talen een datatype werd toegevoegd, probeerde de andere taal dit type ook te integreren. Een voorbeeld van een nieuwe datatype is 'date', om een datum voor te stellen. Ook zijn er een reeks functies gedefinieerd op deze nieuwe types. Een hele krachtige feature is de mogelijkheid om in XSLT een functie te specificeren en die dan op te roepen in een XPath 2.0 expressie.

We zullen nu een aantal nieuwigheden bespreken, waarvan we aantonen dat ze eigenlijk redundant zijn. Hiermee bedoelen we dat deze uitbreidingen van XSLT 2.0 enkel maar zijn toegevoegd voor het gebruikersgemak. Ze maken XSLT dus zeker niet krachtiger.

Start van een transformatie. Net zoals in XSLT 1.0, wordt in de 2.0-versie een transformatie gestart met het zoeken van een template waarvan het bijbehorende patroon de root van het XML-bestand selecteert en waarvan de mode 'start' is. Het is nu echter ook mogelijk om de transformatie te starten door een template met naam *naam* op te roepen. We kunnen dit eenvoudig simuleren in XSLT 1.0. Maak namelijk één template met mode 'start' die de root van de template selecteert. Deze template bevat enkel een call-templatestatementknoop die de template met naam *naam* oproept.

Globale variabelen. De globale variabelen worden niet meer berekend bij de start, maar pas op het moment dat deze nodig blijken te zijn in de omzetting. Deze aanpassing kan een belangrijke invloed hebben op globale variabelen waarvan de berekening oneindig lang duurt. Wordt een dergelijke globale variabele niet gebruikt tijdens de omzetting, is het toch mogelijk dat de transformatie slechts eindige tijd inneemt. De truc om deze feature te simuleren in XSLT 1.0, is om geen globale variabelen te gebruiken. Voor iedere globale variabele v van het type result-tree-fragment wordt een nieuwe template gemaakt, met als mode een unieke naam, bijvoorbeeld 'berekenVariabeleV'. De expressie horende bij de template is 'root'. Op iedere plaats in het programma waar v gebruikt wordt, wordt een nieuwe *locale* variabele v aangemaakt. De berekening van v gebeurt dan als:

```
<xsl:variable name="v">
  <xsl:apply-templates select="/" mode="berekenVariabeleV"/>
</xsl:variable>
```

Merk op dat de berekening van v uitgevoerd wordt met als contextknoop de root van het inputbestand, zoals dat ook zou gebeuren als we v globaal hielden. Voor een variabele van het type node-set of teller, kunnen we gewoon de globale variabeleberekening kopiëren naar iedere plaats waar v gebruikt zal worden. De expressies van de patroontaal moeten we dan wel laten starten vanuit de root, dus niet relatief aan de huidige contextknoop, zodat de huidige context het resultaat niet kan beïnvloeden.

Tunnel parameters. Tunnel parameters zijn een nieuwe soort parameters. Wanneer deze meegegeven worden aan een template, is de waarde van deze parameters gekend in het verdere verloop van de transformatie. Ook deze uitbreiding is redundant. Een template die een verzameling parameters meekrijgt kan gewoon bij iedere aanroep van een andere template (met behulp van call-template of apply-templates) deze verzameling parameters doorgeven.

Het processing model heeft geen grote wijzigingen doorgemaakt. Enkele kleine wijzigingen zijn dat de default template, die geïnstantieerd werd wanneer een expressie bij apply-templates een knoop selecteerde waar geen geschikte regel voor gevonden kon worden, de meegegeven parameters doorgeeft. In voorbeeld 7.1 wordt dit principe uitgelegd. Een template kan nu ook meerdere modes hebben, hoewel het natuurlijk maar een kleine moeite is om dit principe te simuleren met XSLT 1.0.

Terwijl men vroeger één resultaatboom had, kan men nu meerdere bomen creëren. Dit kan handig zijn om vanuit een XML-bestand een site te maken die uit meerdere deelpagina's bestaat.

De belangrijkste wijziging is echter toch het vervangen van het type result-tree-fragment door temporary tree. Net zoals een result-tree-fragment is een temporary tree een forest waaraan, via een variabelebinding, een naam werd gegeven. We kunnen echter deze temporary trees gebruiken in expressies van de patroontaal. Dit houdt in dat de waarde van een dergelijke variabele de uitvoering van een transformatie kan beïnvloeden.

Een volledige lijst van alle wijzigingen ten opzichte van XSLT 1.0 is te vinden in appendix J van de XSLT 2.0 working draft [Kay05].

Voorbeeld 7.1. Beschouw de stylesheet in figuur 7.1. We voeren deze uit volgens de XSLT 1.0-specificatie op de inputboom gegeven in figuur 7.2. Het resultaat is:

```
De waarde van de parameter in d is:
De waarde van de parameter in c is: 5
```

De waarde van v in de laatste regel is dus niet bekend. Voeren we dezelfde stylesheet uit volgens de XSLT 2.0 specificatie, dan krijgen we als resultaat op dezelfde inputboom:

```
De waarde van de parameter in d is: 5
De waarde van de parameter in c is: 5
```

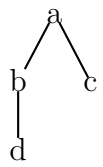
De built-in template voor de knoop b geeft de parameters dus mee door. ■

7.2 Abstracte syntax en formele semantiek voor XSLT 2.0

Uit het voorgaande blijkt dat in XSLT 2.0 een heel uitgebreid gamma aan constructies werd toegevoegd. De meeste van deze wijzigingen zijn ingevoerd

```
<xsl:stylesheet version="x.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:template match="*">
        <xsl:apply-templates select="./*" mode="m">
            <xsl:with-param name="v" select="5"/>
        </xsl:apply-templates>
    </xsl:template>

    <xsl:template match="c" mode="m">
        <xsl:param name="v"/>
            De waarde van de parameter in c is: <xsl:copy-of select="$v"/>
    </xsl:template>

    <xsl:template match="d" mode="m">
        <xsl:param name="v"/>
            De waarde van de parameter in d is: <xsl:copy-of select="$v"/>
    </xsl:template>

</xsl:stylesheet>
```

Figuur 7.1: Voorbeeld built-in templates, verschil tussen 1.0 en 2.0



Figuur 7.2: Inputboom voor voorbeeld 7.1

voor het gemak van de gebruiker. Vanuit theoretisch standpunt is er echter maar één grote wijziging die een enorm effect zal hebben op de complexiteit van XSLT 2.0, namelijk de temporary trees. We zullen hier deze temporary trees inbrengen in de abstracte syntax en formele semantiek van XSLT 2.0.

Omdat de syntax van XSLT 2.0 zeer weinig verschilt met die van XSLT 1.0, zullen we hier enkel de wijzigingen noteren. We baseren ons dus op sectie 4.1. Het enige wat verandert in de definitie van een programma Π, is dat de functie *type* niet langer een variabele kan afbeelden op het type 'result-tree-fragment'. Dit type wordt vervangen door 'temporary tree'. Als we deze wijziging ook doorvoeren in de rest van de syntaxbeschrijving (we vervangen dus overal 'result-tree-fragment' door 'temporary tree'), hebben we de nieuwe syntax van XSLT 2.0 afgewerkt. Syntactisch gezien is dus een temporary tree gewoon een nieuwe naam voor een result-tree-fragment[1].

Syntactisch verandert er dus weinig. Het verschil zit vooral in de semantiek.

De patroontaal $\wp$ beschikt nog steeds over de expressies 'root' en 'children', waarvan de betekenis niet verandert. Het grote verschil is echter dat tussentijdse bomen nu niet langer genegeerd worden. In XSLT 1.0 werd een result-tree-fragment beschouwd als string, waarop geen expressies uit $\wp$ geëvalueerd konden worden. In versie 2.0 worden de result-tree-fragments vervangen door temporary trees, waarop men alle expressies kan evalueren. De functie eval is nu dus gedefinieerd op de volledige context. Deze wijziging heeft een enorm effect op het aantal berekenbare boomtransformaties die met XSLT kunnen uitgevoerd worden. Hierop komen we in hoofdstuk 8 nog uitgebreid op terug.

In de semantiek van XSLT 1.0 zagen we dat indien bij een apply-templatesstatementknoop a geen geldige regel voor een bepaalde knoop n gevonden kon worden, we deze zelf creëerden (zie pg. 75). Deze zogenaamde built-in template gaat op zoek naar geschikte regels voor de kinderen van knoop n. Echter kregen deze regels de parameters die werden gedefinieerd door a niet meer mee. Dit verandert in XSLT 2.0. We passen daarom de alinea in de semantiek van XSLT 1.0 in verband met built-in templates als volgt aan:

Indien nu voor een zekere i tussen 1 en p geen regel gevonden kon worden, creëren we zelf een template, $template_i$. Dit is een boom waarvan de root gelabeld is met 'template'. Het enige kind van deze root is een apply-templatesstatementknoop met als label 'apply-templates', samen met de expressie 'children'. Ook bevat het label de verschillende $(v, expressie)$ paren die voorkwamen in het label van s. Het laatste onderdeel geeft de mode aan, deze

[1]Uit de XSLT 2.0-working draft [Kay05]: The result tree fragment data-type is eliminated. A variable-binding element with content now constructs a temporary tree.

is *mode*. De kinderen van de net gemaakte apply-templatesstatementknoop zijn kopies van de kinderen van s. Dit zijn volges de syntax allemaal with-paramstatementknopen.

Wat is nu de invloed van de temporary trees op de complexiteit? Dit is het onderwerp van het laatste hoofdstuk. Daar zullen we aantonen dat door deze temporary trees XSLT 2.0 Turing compleet is!

Hoofdstuk 8

Computationele volledigheid van XSLT 2.0

In dit onderdeel wordt de Turing-compleetheid van XSLT 2.0 aangetoond. De tactiek bestaat erin de werking van een willekeurige Turing machine te simuleren met behulp van een XSLT-programma. Hiervoor moet dus ook afgesproken worden in welke vorm een Turing machine als input wordt gegeven. De manier die we hier bespreken is uitgevonden door Alexander Korlukov [Kor]. Er wordt in de stylesheet geen gebruik gemaakt van ingebouwde string- of arithmetische functies. Het idee is om alle noodzakelijke informatie over de Turing machine in een speciale (boom)structuur op te slaan, zodanig dat alle gegevens voor het uitvoeren van één stap gemakkelijk te achterhalen zijn.

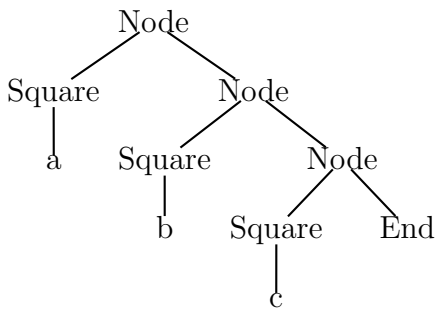
8.1 Simulatie van een Turing machine met XSLT 2.0

Alvorens een Turing machine te definiëren in XML, kijken we eerst hoe we strings en transities kunnen voorstellen. Daarna combineren we deze twee met de huidige toestand, zodat we alle onderdelen van een Turing machine hebben in XML-formaat.

8.1.1 Voorstelling van strings

Het gemakkelijkste om de string-voorstelling uit te leggen, is aan de hand van een voorbeeld.

Voorbeeld 8.1. Beschouw de string `abc`. De boomvoorstelling wordt gegeven in figuur 8.1.



Figuur 8.1: Boomvoorstelling van de string `abc`

De string wordt dus voorgesteld als een binaire boom. Hierin speelt de knoop 'Node' de concatenatie-operator. De extra knoop 'Square' stelt een karakter voor. De naam van deze knoop is misschien op het eerste zicht wat raar, maar later zal ieder karakter de inhoud zijn van één cel op de tape van een Turing machine. Wanneer een 'Node'-knoop als tweede kind de knoop met label 'End' heeft, weten we dat we op het einde zitten van onze string.

Figuur 8.2 geeft de DTD weer voor de bomen die een string voorstellen. ■

```
<!ELEMENT Node ((Square, Node) | (Square, End)) >
<!ELEMENT Square    (#PCDATA) >
```

Figuur 8.2: DTD voor de boomvoorstelling van een string

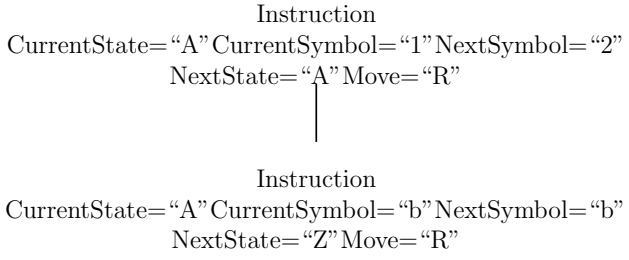
Het voordeel om strings op deze manier op te slaan, is dat het heel eenvoudig is om het eerste symbool van een string te verwijderen. De boomvoorstelling van de string `bc`, dit is de string `abc` zonder het eerste symbool, is simpelweg de boom geworteld in het twee kind van de root van de boomvoorstelling van `abc`.

8.1.2 Voorstelling van transities

Een Turing machine wordt gedefinieerd aan de hand van transities. Deze transities bepalen, aan de hand van de huidige toestand en het huidige symbool onder de lees-en schrijfkop, een nieuwe toestand, een nieuw symbool

en de beweging van de lees- en schrijfkop. Om een transitie voor te stellen, moeten we dus essentieel deze vijf waardes opslaan. De voorstelling van één transitie in de aanpak van Korlukov is een knoop, gelabeld met 'Instruction', met vijf attributen (CurrentState, CurrentSymbol, NextSymbol, NextState, Move). Om alle transities voor te stellen, gebruik Korlukov een lijst van transities.

Voorbeeld 8.2. Beschouw een Turing machine die werkt op strings bestaande uit het symbool '1'. De werking is niets anders dan iedere '1' omzetten in een '2'. Hiervoor gebruiken we twee toestanden en twee transities. De begintoestand noemen we 'A', de eindtoestand 'Z'. Voor een blanco gebruiken we een 'b'. De eerste transtitie verandert bij het lezen van een '1' de huidige cel in een '2', beweegt naar rechts en blijft in toestand 'A'. De andere gaat bij het lezen van een blanco (we zijn dan op het einde van de inputstring) de toestand veranderen naar 'Z'. Als lijst kunnen deze twee transities voorgesteld worden als de boom in figuur 8.3. ■



Figuur 8.3: Boomvoorstelling van twee transities

Tenslotte wordt de DTD voor een dergelijke transitie-boom weergegeven in figuur 8.4.

8.1.3 Voorstelling van een Turing machine

Nu we in staat zijn strings en transities op te slaan, kunnen we een Turing machine creëren. In de stylesheet van Korlukov, die deze machine dan gaat uitvoeren, zijn er wel een aantal belangrijke aannames. Zo moet de stopstate de naam 'Z' dragen en een blancosymbool wordt voorgesteld met 'b'. De string op de tape vóór de lees- en schrijfkop, die altijd moet beginnen met twee blanco's, wordt achterstevoren opgeslagen. De reden hiervoor zal in de werking duidelijk worden. Figuur 8.5 geeft de DTD van een Turing machine.

We hebben dus eerst onze transities. Deze worden opgeslagen in de lijststructuur zoals uitgelegd in sectie 8.1.2. Daarna komt de state waarin we op

```
<!ELEMENT Instruction (Instruction?) >

<!ATTLIST Instruction
    CurrentState  CDATA #REQUIRED
    CurrentSymbol CDATA #REQUIRED
    NextSymbol    CDATA #REQUIRED
    NextState     CDATA #REQUIRED
    Move          CDATA #REQUIRED >
```

Figuur 8.4: DTD voor de boomvoorstelling van transities

dit moment zitten. Wanneer we dus een Turing machine willen opschrijven met behulp van deze DTD, zal deze state de startstate zijn. Daarna komt de tape-inhoud. Deze wordt opgedeeld in drie stukken. Eerst komt de substring van het begin van de tape tot aan de huidige koppositie (TapeLeft genoemd), gevolg door het huidige symbool en daarna de rest van de tape-inhoud (TapeRight). Hierbij is het belangrijk dat TapeLeft de omgekeerde string is van de werkelijke inhoud. Beide strings worden opgeslagen volgens de structuur besproken in sectie 8.1.1.

Voorbeeld 8.3. In hoofdstuk 2 hadden we een voorbeeld van een Turing machine die palindromen over $\{0, 1\}$ herkent (voorbeeld 2.1). We creëren nu de beschrijving van deze Turing machine volgens bovenstaande DTD in figuur 8.6. In deze XML-beschrijving zit ook meteen de input van de Turing machine. De stoptoestand moet de naam 'Z' dragen. We hebben dan het probleem dat de machine wel stopt in Z, maar dan weten we nog niet of dit accept of reject voorstelt. Daarom zullen we op de tape een 'T' schrijven indien het woord geaccepteerd wordt. Een 'F' geeft aan dat de inputstring geen palindroom is. ■

8.1.4 Werking van de stylesheet

De stylesheet die een Turing machine simuleert, krijgt als input een geldig XML-bestand dat voldoet aan de DTD uit figuur 8.5. Het idee is dat iedere stap in de uitvoering een nieuwe Turing machine creëert, net zolang totdat de state vermeld in deze Turing machine 'Z' is. Doordat we al onze belangrijke data (namelijk de lijst van instructies en de verschillende strings) in een speciale structuur hebben opgeslagen, is alle noodzakelijke informatie voor

```
<!ELEMENT TM (Instruction, State, TapeLeft, Symbol, TapeRight)>
<!ELEMENT TapeLeft  (Node) >
<!ELEMENT TapeRight (Node) >
<!ELEMENT Node ((Square, Node) | (Square, End)) >
<!ELEMENT Square    (#PCDATA) >
<!ELEMENT Symbol    (#PCDATA) >
<!ELEMENT State (#PCDATA) >
<!ELEMENT Instruction (Instruction?) >

<!ATTLIST Instruction
    CurrentState  CDATA #REQUIRED
    CurrentSymbol CDATA #REQUIRED
    NextSymbol    CDATA #REQUIRED
    NextState     CDATA #REQUIRED
    Move          CDATA #REQUIRED >
```

Figuur 8.5: DTD voor de boomvoorstelling van een Turing machine

het uitvoeren van één stap onmiddellijk bereikbaar.

- Het eerste wat we zeker nodig hebben, is een instructie die we kunnen toepassen. Wanneer de eerste instructie van onze lijst niet toepasbaar is, kunnen we een (tijdelijke) Turing machine creëren die volledig hetzelfde als de huidige, behalve dat de bovenste instructie weg is. De overige instructies zijn gemakkelijk te bereiken. Het resultaat hiervan is een nieuwe lijst van instructies, die toch nog voldoet aan de DTD. Dit doen we recursief verder totdat een toepasbare transitie gevonden is (we zijn in de veronderstelling dat de Turing machine compleet is, m.a.w. er is een transitie voor iedere mogelijke situatie).

- We moeten rechts bewegen. Er moet een symbool achteraan worden toegevoegd bij het linkse deel van de string op de tape. Omdat deze TapeLeft achterstevoren werd opgeslagen, is dit eenvoudig uit te voeren. Voorts moet het eerste symbool van TapeRight nu het huidige symbool worden. We stellen dan de nieuwe TapeRight gelijk aan het rechterkind van de root van de vorige TapeRight.

- We moeten links bewegen. Dit is analoog aan rechts bewegen, waardoor

```
<TM>

<Instruction CurrentState="Qstart" CurrentSymbol="1" NextSymbol="b" NextState="1gedetecteerd" Move="R">
<Instruction CurrentState="Qstart" CurrentSymbol="0" NextSymbol="b" NextState="0gedetecteerd" Move="R">
<Instruction CurrentState="Qstart" CurrentSymbol="b" NextSymbol="b" NextState="Qaccept" Move="R">

<Instruction CurrentState="1gedetecteerd" CurrentSymbol="0" NextSymbol="0" NextState="1gedetecteerd" Move="R">
<Instruction CurrentState="1gedetecteerd" CurrentSymbol="1" NextSymbol="1" NextState="1gedetecteerd" Move="R">
<Instruction CurrentState="1gedetecteerd" CurrentSymbol="b" NextSymbol="b" NextState="A" Move="L">

<Instruction CurrentState="0gedetecteerd" CurrentSymbol="0" NextSymbol="0" NextState="0gedetecteerd" Move="R">
<Instruction CurrentState="0gedetecteerd" CurrentSymbol="1" NextSymbol="1" NextState="0gedetecteerd" Move="R">
<Instruction CurrentState="0gedetecteerd" CurrentSymbol="b" NextSymbol="b" NextState="B" Move="L">

<Instruction CurrentState="A" CurrentSymbol="1" NextSymbol="b" NextState="C" Move="L">
<Instruction CurrentState="A" CurrentSymbol="b" NextSymbol="b" NextState="Qaccept" Move="L">
<Instruction CurrentState="A" CurrentSymbol="0" NextSymbol="b" NextState="Qreject" Move="L">

<Instruction CurrentState="B" CurrentSymbol="0" NextSymbol="b" NextState="C" Move="L">
<Instruction CurrentState="B" CurrentSymbol="b" NextSymbol="b" NextState="Qaccept" Move="L">
<Instruction CurrentState="B" CurrentSymbol="1" NextSymbol="b" NextState="Qreject" Move="L">

<Instruction CurrentState="C" CurrentSymbol="0" NextSymbol="0" NextState="C" Move="L">
<Instruction CurrentState="C" CurrentSymbol="1" NextSymbol="1" NextState="C" Move="L">
<Instruction CurrentState="C" CurrentSymbol="b" NextSymbol="b" NextState="Qstart" Move="R">


<Instruction CurrentState="Qaccept" CurrentSymbol="b" NextSymbol="T" NextState="Z" Move="R">
<Instruction CurrentState="Qreject" CurrentSymbol="b" NextSymbol="F" NextState="Z" Move="R">
<Instruction CurrentState="Qreject" CurrentSymbol="0" NextSymbol="F" NextState="Z" Move="R">
<Instruction CurrentState="Qreject" CurrentSymbol="1" NextSymbol="F" NextState="Z" Move="R">

</Instruction></Instruction></Instruction></Instruction></Instruction></Instruction></Instruction>
</Instruction></Instruction></Instruction></Instruction></Instruction></Instruction></Instruction>
</Instruction></Instruction></Instruction></Instruction></Instruction></Instruction></Instruction>
</Instruction>

<State>Qstart</State>

<TapeLeft>
<Node><Square>b</Square>
<Node><Square>b</Square>
<End/></Node></Node></TapeLeft>

<Symbol>1</Symbol>

<TapeRight>
<Node><Square>0</Square>
<Node><Square>1</Square>
<Node><Square>0</Square>
<Node><Square>1</Square>
<Node><Square>b</Square>
<Node><Square>b</Square>
<End/></Node></Node></Node></Node></Node></Node></TapeRight>

</TM>
```

Figuur 8.6: Uitgewerkte Turing machine die palindromen aanvaardt

we weer volop profiteren van de speciale string-structuur.

8.2 Berekenbare boomtransformaties en XSLT 2.0

We weten dat we iedere boom met labels uit een alfabet Σ kunnen bekijken als een string over het alfabet $\Sigma \cup \{'(', ')'\}$. De nesting van een boom wordt nu voorgesteld met behulp van haakjes. Als gevolg daarvan kunnen we boomtransformaties ook bekijken als stringtransformaties. Een stringtransformatie wordt *berekenbaar* genoemd als ze berekend kan worden met een Turing machine. We breiden deze definitie van berekenbaarheid nu op een triviale manier uit naar boomtransformaties.

definitie 8.1. Een boomtransformatie wordt berekenbaar genoemd als ze, bekeken als stringtransformatie, berekenbaar is door een Turing machine.

In de vorige sectie hebben we aangetoond dat het mogelijk is met XSLT 2.0 een willekeurige Turing machine te simuleren. Hiermee is echter nog niet aangetoond dat iedere berekenbare boomtransformatie uitvoerbaar is met XSLT 2.0. Een boomtransformatie is berekenbaar indien ze uitvoerbaar is op een Turing machine. Hiervoor moeten we de boom wel eerst in een stringformaat omzetten.

stelling 8.1. Iedere berekenbare boomtransformatie is berekenbaar met XSLT 2.0.

Bewijs. Zij P een berekenbare boomtransformatie en t een boom over het eindige alfabet Σ. Zij s de stringvoorstelling van t en s' de stringvoorstelling van $P(t)$, waarbij we de stringvoorstelling van sectie 8.1.1 gebruiken. We moeten aantonen dat $P(t)$ berekenbaar is met behulp van XSLT 2.0. De definitie van een berekenbare boomtransformatie is dat de s omgezet kan worden in s', met behulp van een Turing machine. Het bewijs verloopt in drie stappen.

1. t kan omgezet worden in s met behulp van XSLT 2.0

   We moeten de boom t omzetten in zijn stringvoorstelling, s. Hiervoor is een stylesheet ontwikkeld, voor $\Sigma = \{a, b, c\}$, wat meteen bewijst dat we een willekeurige boom over Σ kunnen omzetten naar zijn stringvoostelling (zie figuren 8.7 en 8.8).

2. s kan omgezet worden in s' met behulp van XSLT 2.0

We moeten nu s omzetten in s'. Omdat P een berekenbare boomtransformatie is, kan een Turing machine deze operatie uitvoeren. We kunnen iedere Turing machine simuleren met behulp van XSLT 2.0. Bijgevolg kunnen we ook s omzetten in s' met behulp van XSLT 2.0.

3. s' kan omgezet worden in $P(t)$ met behulp van XSLT 2.0

   We moeten een stringvoorstelling van een boom terug omzetten in een boom. Ook hiervoor werd een stylesheet ontwikkeld (zie figuren 8.9, 8.10 en 8.11), waarmee we aantonen dat we een willekeure stringvoorstelling van een boom kunnen omzetten in de boom zelf.

□

8.3 Besluit

Een belangrijk besluit dat we kunnen trekken uit dit hoofdstuk is het feit dat de ontwikkelaars van XSLT 2.0 hun doelstellingen niet zijn nagekomen. In deze 'XSLT Requirements 2.0' [MMS01] stond vermeld dat het geen doel is om XSLT 2.0 om te vormen in een general purpose taal. Korlukov toonde aan dat dit echter wel het geval is.

Waarom was de tactiek van hoofdstuk 5 (terminatie van XSLT 1.0) niet bruikbaar voor XSLT 2.0? Bij terminatie van XSLT 1.0 konden we variabelen van het type result-tree-fragment negeren, omdat hun waarde geen enkele invloed kon uitoefenen op de uitvoering van een stylesheet. Doordat temporary trees wel gebruikt kunnen worden in een patroon, kan de uitwerking wel beïnvloed worden. We konden bij het zoeken naar een herhaling de temporary trees wel in rekening brengen. Echter biedt dit geen oplossing: we kunnen namelijk geen grens zetten op het aantal temporary trees. Dit ging duidelijk wel met tellers en node-sets.

```
<xsl:stylesheet version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:template match="/">
        <xsl:variable name="broad">
            <S>
                <xsl:apply-templates select="." mode="makeBroadRepresentation"/>
            </S>
        </xsl:variable>

        <xsl:apply-templates select="$broad" mode="toStringRepresentation"/>
    </xsl:template>

    <xsl:template match="*" mode="toStringRepresentation">
        <xsl:variable name="newParameter">
            <S>
                <xsl:copy-of select="./*[position() > 1]"/>
            </S>
        </xsl:variable>

        <xsl:choose>

            <xsl:when test="count(./*) = 1">
                <Node>
                    <xsl:copy-of select="./*"/>
                    <xsl:call-template name="makeEnd"/>
                </Node>
            </xsl:when>

            <xsl:otherwise>
                <Node>
                    <xsl:copy-of select="./*[position() = 1]"/>
                    <xsl:apply-templates select="$newParameter" mode="toStringRepresentation"/>
                </Node>
            </xsl:otherwise>
        </xsl:choose>
    </xsl:template>

    <!--zet de inputboom om in een brede voorstelling-->
    <xsl:template match="*" mode="makeBroadRepresentation">
        <xsl:choose>
            <!--we zitten NIET in een blad!-->
            <xsl:when test="./*">
                <xsl:call-template name="makeSquare"/>
                <xsl:call-template name="makeOpen"/>
                    <xsl:apply-templates select="./*" mode="makeBroadRepresentation"/>
                <xsl:call-template name="makeClose"/>
            </xsl:when>
```

Figuur 8.7: Stylesheet: van boom naar stringvoorstelling, deel 1

```
            <!--we zitten in een blad!-->
            <xsl:otherwise>
                <xsl:call-template name="makeSquare"/>
            </xsl:otherwise>
        </xsl:choose>
    </xsl:template>

    <!--maak een Square element van een openhaakje-->
    <xsl:template name="makeOpen">
        <Square>
            <open/>
        </Square>
    </xsl:template>

    <!--maak een Square element van een sluithaakje-->
    <xsl:template name="makeClose">
        <Square>
            <close/>
        </Square>
    </xsl:template>

    <!--maak een Square element van de huidige contentnode-->
    <xsl:template name="makeSquare">
        <xsl:choose>
            <xsl:when test="self::a">
                <Square>
                    <a/>
                </Square>
            </xsl:when>
            <xsl:when test="self::b">
                <Square>
                    <b/>
                </Square>
            </xsl:when>
            <xsl:when test="self::c">
                <Square>
                    <c/>
                </Square>
            </xsl:when>
        </xsl:choose>
    </xsl:template>

    <!--maak een end element-->
    <xsl:template name="makeEnd">
        <end/>
    </xsl:template>

</xsl:stylesheet>
```

Figuur 8.8: Stylesheet: van boom naar stringvoorstelling, deel 2

```
<xsl:stylesheet version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:template match="Node">

        <xsl:variable name="token">
            <xsl:value-of select="name(./Square[1]/*)"/>
        </xsl:variable>

        <xsl:variable name="nextToken">
            <xsl:value-of select="name(./Node/Square[1]/*)"/>
        </xsl:variable>

        <xsl:choose>
            <xsl:when test="$token = 'end'">
                <!-- we zijn aan het einde-->
            </xsl:when>

            <xsl:when test="$token = 'close'">
                <!-- we zijn aan het einde-->
            </xsl:when>

            <xsl:when test="$nextToken = 'open'">

                <xsl:variable name="diepteAchterHaakje">
                    <xsl:apply-templates select="./Node/Node" mode="zoekSluitHaakje">
                        <xsl:with-param name="aantalOpen" select="1"/>
                    </xsl:apply-templates>
                </xsl:variable>

                <xsl:variable name="diepte">
                    <xsl:apply-templates select="./Node/Node" mode="bepaalDiepte"/>
                </xsl:variable>

                <xsl:variable name="plaatsHaakje">
                    <xsl:value-of select="$diepte - $diepteAchterHaakje"/>
                </xsl:variable>

                <xsl:variable name="stukVoorSluitHaakje">
                    <xsl:apply-templates select="./Node/Node" mode="deelVoorHaakje">
                        <xsl:with-param name="plaatsHaakje" select="$plaatsHaakje"/>
                    </xsl:apply-templates>
                </xsl:variable>

                <xsl:variable name="stukNaSluitHaakje">
                    <xsl:apply-templates select="./Node/Node" mode="deelNaHaakje">
                        <xsl:with-param name="plaatsHaakje" select="$plaatsHaakje"/>
                    </xsl:apply-templates>
                </xsl:variable>

                <xsl:element name="$token">
                    <xsl:apply-templates select="$stukVoorSluitHaakje"/>
                </xsl:element>
```

Figuur 8.9: Stylesheet: van stringvoorstelling naar boom, deel 1

```
            <xsl:apply-templates select="$stukNaSluitHaakje"/>

        </xsl:when>

        <xsl:otherwise>
            <xsl:element name="$token"/>
            <xsl:apply-templates select="./Node"/>
        </xsl:otherwise>
    </xsl:choose>

</xsl:template>

<xsl:template match="node()" mode="deelVoorHaakje">
    <xsl:param name="plaatsHaakje"/>

    <xsl:choose>
        <xsl:when test="$plaatsHaakje = 0">
        </xsl:when>

        <xsl:otherwise>
            <Node>
                <xsl:copy-of select="./*[1]"/>

                <xsl:apply-templates select="./*[2]"  mode="deelVoorHaakje">
                    <xsl:with-param name="plaatsHaakje" select="$plaatsHaakje - 1"/>
                </xsl:apply-templates>
            </Node>
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>

<xsl:template match="node()" mode="deelNaHaakje">
    <xsl:param name="plaatsHaakje"/>

    <xsl:choose>
        <xsl:when test="$plaatsHaakje = 0">
            <xsl:copy-of select="."/>
        </xsl:when>

        <xsl:otherwise>
            <xsl:apply-templates select="./*[2]"  mode="deelNaHaakje">
                <xsl:with-param name="plaatsHaakje" select="$plaatsHaakje - 1"/>
            </xsl:apply-templates>
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>
```

Figuur 8.10: Stylesheet: van stringvoorstelling naar boom, deel 2

```
    <xsl:template match="node()" mode="zoekSluitHaakje">
        <xsl:param name="aantalOpen"/>

        <xsl:variable name="token" select="name(./Square[position() = 1]/*)"/>

        <xsl:choose>

            <xsl:when test="count(./Node/*) = 0">
                <xsl:value-of select="0"/>
            </xsl:when>

            <xsl:when test="$aantalOpen = 0">
                <xsl:apply-templates select="." mode="bepaalDiepte"/>
            </xsl:when>

            <xsl:when test="$token = 'open'">
                <xsl:apply-templates select="./Node" mode="zoekSluitHaakje">
                    <xsl:with-param name="aantalOpen" select="$aantalOpen + 1"/>
                </xsl:apply-templates>
            </xsl:when>

            <xsl:when test="$token = 'close'">
                <xsl:apply-templates select="./Node" mode="zoekSluitHaakje">
                    <xsl:with-param name="aantalOpen" select="$aantalOpen - 1"/>
                </xsl:apply-templates>
            </xsl:when>

            <xsl:otherwise>
                <xsl:apply-templates select="./Node" mode="zoekSluitHaakje">
                    <xsl:with-param name="aantalOpen" select="$aantalOpen"/>
                </xsl:apply-templates>
            </xsl:otherwise>
        </xsl:choose>

    </xsl:template>

    <xsl:template match="node()" mode="bepaalDiepte">
        <xsl:param name="diepte" select="1"/>

        <xsl:choose>
            <xsl:when test="count(./*[2]) = 0">
                <xsl:value-of select="$diepte"/>
            </xsl:when>

            <xsl:otherwise>
                <xsl:apply-templates select="./*[2]" mode="bepaalDiepte">
                    <xsl:with-param name="diepte" select="$diepte + 1"/>
                </xsl:apply-templates>
            </xsl:otherwise>
        </xsl:choose>
    </xsl:template>


</xsl:stylesheet>
```

Figuur 8.11: Stylesheet: van stringvoorstelling naar boom, deel 3

Bibliografie

[Bak78] Brenda S. Baker. Generalized syntax directed translation, tree tranducers, and linear space. *Siam Journal on Computing*, 7(3):376–391, augustus 1978.

[BBC+05] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. *XML Path Language, Xpath 2.0*, 4 april 2005. http://www.w3.org/TR/xpath20/.

[Ber] Tim Berners-Lee. *World Wide Web Consortium.* http://www.w3.org.

[BMN02] Geert Jan Bex, Sebastian Maneth, and Frank Neven. A formal model for an expressive fragment of xslt. *Inf. Syst.*, 27(1):21–39, 2002, doi:http://dx.doi.org/10.1016/S0306-4379(01)00033-3.

[Bur01] Eric M. Burke. *Java and XSLT.* O'reilly, september 2001.

[CD99] James Clark and Steven DeRose. *XML Path Language, Xpath 1.0*, 16 november 1999. http://www.w3.org/TR/xpath.

[CDG+97] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata, 1997. release October, 1rst 2002.

[Cla99] James Clark. *XSL Transformations (XSLT), Version 1.0*, 16 november 1999. http://www.w3.org/TR/xslt.

[Cla01] James Clark. *XSL Transformations (XSLT), Version 1.1, W3C Working Draft*, 24 augustus 2001. http://www.w3.org/TR/2001/WD-xslt11-20010824/.

[DM79] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Commun. ACM*, 22(8):465–476, 1979, doi:http://doi.acm.org/10.1145/359138.359142.

[Eng86] Joost Englfriet. The complexity of languages generated by attribute grammars. *SIAM J. Comput.*, 15(1):70–86, 1986, doi:http://dx.doi.org/10.1137/0215005.

[Fit03] Michael Fitzgerald. *Learning XSLT*. O'reilly, november 2003.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, Mass., 1979.

[Hue80] Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. ACM*, 27(4):797–821, 1980, doi:http://doi.acm.org/10.1145/322217.322230.

[Kaya] Michael Kay. *saxon xslt-processor*. http://saxon.sourceforge.net/.

[Kayb] Michael Kay. *saxon xslt-processor, betaalversie*. http://www.saxonica.com/.

[Kay01] Michael Kay. *XSLT*. Wrox Press Ltd, 2 edition, 2001.

[Kay05] Michael Kay. *XSL Transformations (XSLT), Version 2.0, W3C Working Draft*, 4 april 2005. http://www.w3.org/TR/2005/WD-xslt20-20050404/.

[KD04] Helmut Kopka and Patrick W. Daily. *Guide to Latex*. Addison-Wesley, 4 edition, 2004.

[Kep04] Stephan Kepser. A simple proof for the turing-completeness of xslt and xquery. *Extreme Markup Languages*, augustus 2004.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[Kor] Alexander Korlukov. Turing machine simulation with xslt 2.0. http://www.refal.net/~korlukov/tm/.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language Second Edition*. Prentice-Hall, Inc., 1988.

[Lam99] Leslie Lamport. *Latex, A Document Preparation System*. Addison-Wesley, second edition, 1999.

[LS77] Robert J. Lipton and Lawrence Snyder. On the halting of tree replacement systems. Research Report 99, Yale University Department of Computer Science, New Haven, CT, 1977.

[Mar02] Michael Marte. A modular approach to proving confluence. In *FroCos*, pages 33–48, 2002.

[MMS01] Steve Muenchand Mark Scardina. *XSLT Requirements Version 2.0*, 14 February 2001. http://www.w3.org/TR/2001/WD-xslt20req-20010214.

[Nev02] Frank Neven. On the power of walking for querying tree-structured data. In Popa [Pop02], pages 77–84.

[Pet91] Alberto Pettorossi. A property which guarantees termination in weak combinatory logic and subtree replacement systems. *Notre Dame Journal of Formal Logic*, 22(4):344–356, October 1991.

[Pop02] Lucian Popa, editor. *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*. ACM, 2002.

[Ros73] Barry K. Rosen. Tree-manipulating systems and church-rosser theorems. *J. ACM*, 20(1):160–187, 1973, doi:http://doi.acm.org/10.1145/321738.321750.

[SG95] C.M. Sperberg-McQueen and Robert F. Goldstein. A manifesto for adding sgml intelligence to the world-wide web. *Computer Networks and ISDN Systems*, 28:3–11, 1995.

[Sip96] M. Sipser. *Introduction to the Theory of Computation*. PWS, Boston, MA, 1996.

[SK95] Ken Slonneger and Barry L. Kurtz. *Formal Syntax and Semantics of Programming Languages: A laboratory based approach*. Addison-Wesley Publishing Company, 1995.

[ST00] C. M. Sperberg-McQueen and Henry Thompson. *XML Schema*, april 2000. http://www.w3.org/XML/Schema.

[Ter03] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[Tid01] Doug Tidwell. *XSLT*. O'Reilly Associates, 1 edition, 2001.

[vL90] Jan van Leeuwen, editor. *Handbook of theoretical computer science (vol. A): algorithms and complexity*. MIT Press, Cambridge, MA, USA, 1990.

[YBP+04] Franois Yergeau, Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. *Extensible Markup Language 1.0*, third edition, februari 2004. http://www.w3.org/TR/2004/REC-xml-20040204/.