Automaten en Berekenbaarheid

3° Bachelor Informatica

Diverse Minoren en Kennisdomeinen

19 november 2013

B. Demoen
KU Leuven
Dep. Computerwetenschappen

Inhoudsopgave

Hoofdstuk 1

Voorwoord

Informatici hebben veel te maken met programma's, en programma's berekenen dikwijls bij een gegeven input een output. Dat moet heel ruim opgevat worden: input kan een elektrisch signaal zijn dat een temperatuursschommeling aanduidt, een karakter dat van het toetsenbord komt, een programma, een gegevensbank ... en de bijbehorende output kan dan een rij signalen zijn om klepopeningen te regelen, een aangepast document, een bestand met foutenboodschappen, een display met gevraagde gegevens ...

Vanuit praktisch standpunt zijn bovenstaande voorbeelden heel verschillend en als we te veel naar de details van de voorbeelden kijken, dan missen we veel, want er is een kader dat abstractie van vele details maakt en waarin elk van die voorbeelden te beschrijven valt: het gaat steeds om een mapping van elementen die uit een inputdomein komen, naar een element uit een outputdomein. Die domeinen zijn altijd discreet, maar soms wel onbegrensd groot. Vanuit theoretisch standpunt zijn programma's dus altijd terug te brengen tot functies van (een deel van) $\mathbb{N}$ naar (een deel van) $\mathbb{N}$.

Als het inputdomein klein is, dan is de functie niet erg interessant: je kan een tabel aanleggen en de functiewaarde voor een gegeven input berekenen is een simpele lookup.[1]

Als het inputdomein groot is, of niet a priori bekend, dan is het voordien tabuleren van de functie niet mogelijk, en is het beter van te doen alsof het inputdomein oneindig groot is.

Het outputdomein moet minstens twee elementen bevatten om een interessante

[1] Tabulatie is in deze context een interessante dynamische implementatietechniek, zowel in functioneel (inclusief OO) als in logisch programmeren (waar relaties worden gespecifieerd i.p.v. functies). Wil je meer weten, vraag !

functie te verkrijgen. Die twee elementen zouden *ja, nee* kunnen zijn en laten toe om *beslissingsproblemen* te beschrijven: problemen waarbij je wil weten of een input voldoet aan een bepaalde eigenschap, zoals *is dit programma syntactisch correct?*, of nog *is vandaag een goede dag om te beleggen in A&B?* [2] ...

Als het outputdomein meer dan twee, maar wel eindig veel elementen bevat, dan is het gemakkelijk om door een rij ja-nee vragen over de input, de juiste output te verkrijgen. Stel dat je wil berekenen op welke dag van de week je best in A&B belegt, dan stel je de volgende ja-nee vragen: *is maandag de beste dag om in A&B te beleggen ?*, *is dinsdag de beste dag om in A&B te beleggen?*, *is woensdag de beste dag om in A&B te beleggen?*, ... Van de eindige outputdomeinen, zijn dus alleen die met juist twee waarden echt interessant.

Als het outputdomein oneindig groot is, dan kunnen we dat niet zomaar reduceren naar een eindige rij functies met eindig outputdomein.

We besluiten dat er maar twee soorten interessante functies bestaan: functies met signatuur $\mathbb{N} \to \{ja, nee\}$ en functies met signatuur $\mathbb{N} \to \mathbb{N}$.

Laat ons nog eens naar die eerste klasse kijken: er is een zekere structuur en semantiek geassocieerd met het outputdomein, maar het inputdomein is zeer generisch. $\mathbb{N}$ staat immers voor een willekeurige aftelbare oneindige verzameling: een andere aftelbare oneindige verzameling zou net zo goed kunnen dienen. Verder: als we elementen van $\mathbb{N}$ noteren, dan gebruiken we (bijvoorbeeld) de tien decimale cijfers en vormen daarmee strings (rijen van decimale cijfers), maar we hadden net zo goed in een andere basis kunnen werken, bijvoorbeeld binair, of hexadecimaal, of op nog totaal andere manier de elementen van $\mathbb{N}$ voorstellen: excess-3, ... Die voorstellingswijzen hebben gemeen dat ze gebruik maken van een eindig aantal symbolen, dat de voorstelling uniek is en dat als je twee voorstellingen (van twee gelijke of verschillende) getallen achter elkaar plaatst, je de voorstelling krijgt van een nieuw getal. De eindige verzameling van symbolen noemen we een alfabet, en gelijk welke eindige rij symbolen stelt een getal voor. Stel met Σ een eindig alfabet voor, en met Σ^* alle eindige rijen van symbolen uit Σ. Dan hebben we nu juist zowat verantwoord dat we functies zullen bekijken met signatuur $\Sigma^* \to \{ja, nee\}$. Een functie F van die klasse wordt nu helemaal bepaald door de verzameling symboolrijen s waarvoor geldt dat $F(s) = ja$ en we kunnen i.p.v. zulke functies te bekijken, net zo goed naar deelverzameling V van Σ^* kijken met de bijbehorende vraag *behoort een gegeven s tot V*. Het is

[2] Ja !

belangrijk om te beseffen dat we geen echt grotere klasse van functies beschouwen dan $\mathbb{N} \to \{ja, nee\}$, maar ook dat het dikwijls beter is om beslissingsproblemen op meer natuurlijke manier te beschrijven, t.t.z. vanuit het standpunt van delen van Σ^*. Bijna alle theorie i.v.m. berekenbaarheid zal geformuleerd worden in dit kader.

De tweede klasse functies met signatuur $(\mathbb{N} \to \mathbb{N})$ zullen iets minder aan bod komen, maar ze zijn natuurlijk heel belangrijk.

Zelf doen:

> De inleiding maakt vereenvoudigingen of veronderstellingen waarmee je misschien niet akkoord gaat, of die op zijn minst argumentatie vereisen. Zoek mogelijke *gaten* in de inleiding, bedenk alternatieven, argumenteer voor en tegen ...
>
> Beschrijf een beslissingsprobleem dat je al kende met behulp van een alfabet en een deelverzameling die de oplossing van het probleem is. Doe dat voor een probleem met cijfers en getallen, een probleem met letters en woorden, ... Zorg dat je telkens een probleem kiest waarvoor de deelverzameling eindig is, en eentje waarvoor de deelverzameling oneindig is. Wat als je ja/nee omkeert?
>
> Blijf tijdens de hele cursus alert: krijg je een aanval van constructivitis, relevantitis, historititis of curiositis, mail de lesgever of onderbreek de les.

Vele fouten, onduidelijkheden en andere onvolkomenheden werden door lezers gemeld en daardoor is deze nieuwe versie heel wat beter dan de eerste.

Oprechte dank aan (in alfabetische orde) Sofie Burggraeve, Philippe De Croock, Lynn Houthuys, Andres Humberto Gutierrez Gonzalez, Joachim Jansen, Lore Kesteloot, Ruth Nysen, Jan Ramon, Peter Roelants, Raoul Strackx, Steven Van Essche, Christophe Van Ginneken, Dieter van Melkebeek, Nick Vannieuwenhoven, Timon Van Overveldt, Pieter Van Riet, Stijn Vermeeren, Peter Verraedt, Olivier Verriest, Sarah Wauters en Johan Wittocx.
Je kan nog steeds problemen met de tekst melden per e-mail.

Hoofdstuk 2

Talen en Automaten

1 Wat is een taal?

De inleiding motiveert om beslissingsproblemen te bekijken. Een beslissingsprobleem komt overeen met een deelverzameling van de rijen (ook strings genoemd) die je kan maken met elementen uit een alfabet Σ. Zulk een deelverzameling noemen we een *taal* over Σ. Elementen uit de taal noemen we *woorden* of *strings*. Een alfabet heeft altijd een eindig aantal elementen.

Definitie 1.1 String over een alfabet Σ
Een **string** over een alfabet Σ is een opeenvolging van nul, één of meer elementen van Σ

Het is duidelijk dat als we twee strings x en y achter elkaar zetten, we een nieuwe string krijgen. We noteren die met xy.

Definitie 1.2 Taal L over een alfabet Σ
Een **taal** L over een alfabet Σ is een verzameling van eindige strings over Σ

Een taal kan eindig zijn of oneindig. Als een taal L oneindig is, is L dan aftelbaar?[1]
Om een taal vast te leggen moet je een beschrijving geven van elk element van de taal. Bijvoorbeeld: de taal van even getallen (over een alfabet van decimale cijfers) bevat juist alle getallen die eindigen op een 0, 2, 4, 6 of 8. Een ander voorbeeld: elk woord dat rijmt op fantastisch. Of nog: elke ...
Een beschrijving van elk element van een taal is liefst eindig, zelfs als de taal oneindig is.

[1] als je niet meer weet wat aftelbaar is, zoek het op

Een beschrijving van elk element van een taal kan je (meestal) gebruiken om na te gaan of een string tot de taal behoort, maar (meestal) ook om elk element van de taal te construeren of genereren. Let hier goed op de *(meestal)*: er zullen later vragen over komen :-)
Als de beschrijving van een taal eenvoudig is, dan verwachten we dat de taal eenvoudig is - maar hebben we wel een goed beeld van wat eenvoudig is?
Hier zijn nog wat vragen om je over te bezinnen en waarop in deze cursus antwoorden worden aangereikt:

- Bestaat er een formalisme om taalbeschrijvingen te noteren?
- Geeft dat formalisme aanleiding tot het (automatisch) afleiden van testers en generators van de taal?
- Bestaan er talen die niet in een formalisme kunnen gevat worden?
- Wat is de goede notie van testen? En genereren?
- Zijn sommige talen inherent gemakkelijker dan andere om te beschrijven/testen/genereren?

Tenslotte is er de vraag:

> Waarom moet een universitaire informaticus dit kennen?

Zelf doen:

> Kan je een beschrijving van de even getallen gebruiken om alle even getallen te genereren?
>
> Alle woorden rijmend op fantastisch?
>
> Hoe zit het met testen?
>
> Verzin zelf een taal, een beschrijving van die taal en gebruik die beschrijving om te testen of een gegeven string tot de taal behoort, en ook om alle strings van de taal te genereren. Hoe extravaganter de taal is, hoe beter.
>
> Zie je in je voorbeelden een verband tussen hoe eenvoudig het is om je taal te beschrijven en testen of te genereren?
>
> Heb je al een gevoel voor de andere vragen die hiervoor gesteld werden?

2 Een algebra van talen

Een algebra - of algebraische structuur - is een verzameling met daarop een aantal inwendige operaties: dikwijls binaire operaties, maar unair of met grotere ariteit kan ook. Zo wordt de verzameling van alle talen over een alfabet Σ een algebra als we als operaties unie, doorsnede, complement ... definiëren. Meer concreet: als L_1 en L_2 twee talen zijn, dan is

- de unie ervan een taal: $L_1 \cup L_2$
- de doorsnede ervan een taal: $L_1 \cap L_2$
- het complement ervan een taal: $\overline{L_1}$

Daarmee kan je nog andere operaties maken.
Een nieuwe manier om uit twee talen een taal te maken is *concatenatie*:

Definitie 2.1 Concatenatie van twee talen
Gegeven twee talen L_1 en L_2 over hetzelfde alfabet Σ, dan noteren we de concatenatie van L_1 en L_2 als L_1L_2 en definiëren we:
$L_1L_2 = \{xy | x \in L_1, y \in L_2\}$

We hebben geen haakjes gezet rond de concatenatie van talen, want het is duidelijk dat concatenatie associatief is, t.t.z. $(L_1L_2)L_3 = L_1(L_2L_3)$
Als we L n keer concateneren met zichzelf, noteren we dat door L^n. L^0 bevat alleen de lege string die we noteren door ϵ.
Tenslotte definiëren we nog een operatie die toelaat om oneindige talen te construeren vanuit een eindige taal:

Definitie 2.2 L^* - de Kleene **ster** van een taal L

$$L^* = \cup_{n=0}^{\infty} L^n$$

Als afkorting voor LL^* wordt L^+ gebruikt.
Met die notatie kunnen we nu een taal L definiëren als een deelverzameling van Σ^*, of equivalent daarmee $L \in \mathcal{P}(\Sigma^*)$

Zelf doen: zoek nieuwe operaties die van een taal een (mogelijke nieuwe) taal maken.

3 Talen beschrijven

In de wiskunde gebruikt men verzamelingennotatie om verzamelingen te beschrijven. Bijvoorbeeld:

Voorbeeld *Met* $\Sigma = \{x, y, z\}$*:*

- $\Sigma^* = \{a_1a_2a_3...a_n | a_i \in \Sigma, n \in \mathbb{N}\}$
- $L = \{a_1a_2a_3...a_n | a_1 = y, n \in \mathbb{N}, \forall i > 1 : a_i \in \Sigma\}$

  L is de verzameling van strings die met y beginnen.

Ook informele beschrijvingen kunnen, zolang ze maar ondubbelzinnig zijn zodat elk *ding* een element is of niet:

Voorbeeld

- $H(n) = \{P | P$ *is een Javaprogramma en P stopt na hoogstens n seconden bij input* $n\}$
- $Prime = \{n | n \in \mathbb{N}, n$ *is een priemgetal*$\}$

Die laatste is ook informeel, maar er zit natuurlijk een definitie van priemgetal achter die formeel kan uitgeschreven worden. Zoiets als

$$Prime = \{n | n \in \mathbb{N}, \forall i : 1 < i < n \rightarrow n \ mod \ i \neq 0\}$$

Voor ons is zulke beschrijving echter niet genoeg: wij willen een formalisme dat beter toelaat om strings te genereren en te testen. Dat bestaat al voor natuurlijke talen: een grammatica voor het nederlands beschrijft de structuur van een nederlandse zin. Nederlands is echter een ingewikkelde taal, met een complexe structuur en veel uitzonderingen: het heeft zin om eerst een klasse meer eenvoudige talen te bestuderen.
We werken toe naar een

hiërarchie van talen

met bijbehorende hiërarchie van beschrijvingsmechanismen of grammatica's

met bijbehorende hiërarchie van *test* en *generatie* procedures

Die hiërarchie heet de *Chomsky-hiërarchie*[2]. We beginnen onderaan, t.t.z. bij de *gemakkelijke* talen ...

[2]Noam Chomsky

4 Reguliere expressies en reguliere talen

Definitie 4.1 Reguliere Expressie (RE) over een alfabet Σ
E is een **reguliere expressie over alfabet** Σ indien E van de vorm is

- ϵ
- ϕ
- a waarbij $a \in \Sigma$
- $(E_1 E_2)$ waarbij E_1 en E_2 reguliere expressies zijn over Σ
- $(E_1)^*$ waarbij E_1 een reguliere expressie is over Σ
- $(E_1|E_2)$ waarbij E_1 en E_2 reguliere expressies zijn over Σ

Hierboven is de verzameling van reguliere expressies *RegExps* op inductieve manier gedefinieerd. Er is onder verstaan dat iets dat niet onder de definitie valt, geen reguliere expressie is. Dikwijls is het impliciet duidelijk welk alfabet we gebruiken en vermelden we het niet meer. We gebruiken haakjes zodat er geen enkele ambiguïteit kan bestaan: haakjes behoren niet tot het alfabet. In de volgende voorbeelden is $\Sigma = \{a, b, c\}$.

Voorbeeld *Reguliere expressies over* Σ*:*

- b
- $a(\epsilon c)$
- $(((ab))^* c \mid (bc))$

Gebruiken we te veel haakjes - of te weinig? Waarom?

Definitie 4.2 Een reguliere expressie E **bepaalt** een taal L_E over hetzelfde alfabet Σ als volgt:

- als E = a (met $a \in \Sigma$) dan is $L_E = \{a\}$ (de taal met één string die enkel het teken a is)
- als E = ϵ dan is $L_E = \{\epsilon\}$ (de lege string)
- als E = ϕ dan is $L_E = \emptyset$ (de lege verzameling)
- als E = $(E_1 E_2)$ dan $L_E = L_{E_1} L_{E_2}$
- als E = $(E_1)^*$ dan $L_E = L^*_{E_1}$
- als E = $(E_1|E_2)$ dan $L_E = L_{E_1} \cup L_{E_2}$

Die overeenkomst tussen een reguliere expressie en een taal maakt dat we in reguliere expressies ook wegkomen met minder haakjes: concatenatie van talen is immers associatief en als we afspreken dat de * sterker bindt dan concatenatie die sterker bindt dan unie, dan kunnen veel haakjes weg.

Zelf doen: Geef een woordelijke beschrijving van de talen die bepaald worden door de volgende RE's over $\{a, b\}$ - de eerste dient als voorbeeld:

$(ab)^*$: elke a wordt direct door een b gevolgd en er zijn evenveel a's als b's

$(aba)^*$

$(a|b)^*$

$(a|b)^*\phi$

$a\epsilon b$

Zelf doen: Bewijs de volgende uitspraken, of geef een tegenvoorbeeld:

als een reguliere expressie E geen * bevat, dan is L_E eindig

als een reguliere expressie E * bevat, dan is L_E oneindig

$L_E \subseteq L_{(E|F)}$ voor alle RE's E en F

de verzameling van alle reguliere expressies (over een gegeven alfabet) is zelf een taal (en over welk alfabet?)

de verzameling van alle reguliere expressies (over een gegeven alfabet) is zelf een reguliere taal

Definitie 4.3 Reguliere Taal
Een taal die door een reguliere expressie bepaald wordt is een **reguliere taal**.

Is het duidelijk dat een reguliere taal een taal is? De verzameling van reguliere talen duiden we aan met $RegLan$. Kijk na welke van volgende formules zin hebben, en welke juist zijn.

1. $RegLan \subseteq \Sigma$
2. $RegLan \subseteq \Sigma^*$
3. $RegLan \subseteq \mathcal{P}(\Sigma)$
4. $RegLan \subseteq \mathcal{P}(\Sigma^*)$
5. $RegLan \subseteq \mathcal{P}(\mathcal{P}(\Sigma^*))$
6. indien $x \in RegLan$ dan $x \in \Sigma$
7. indien $x \in RegLan$ dan $x \in \Sigma^*$
8. indien $x \in RegLan$ dan $x \in \mathcal{P}(\Sigma)$
9. indien $x \in RegLan$ dan $x \in \mathcal{P}(\Sigma^*)$
10. indien $x \in RegLan$ en $y \in x$ dan $y \in \Sigma$
11. indien $x \in RegLan$ en $y \in x$ dan $y \in \Sigma^*$
12. indien $x \in RegLan$ en $y \in x$ dan $y \in \mathcal{P}(\Sigma)$
13. indien $x \in RegLan$ en $y \in x$ dan $y \in \mathcal{P}(\Sigma^*)$

Zelf doen:

Is de volgende uitspraak juist? *voor elke reguliere taal L bestaat een reguliere expressie E zodanig dat* $L_E = L$.

Is het duidelijk dat er talen zijn die NIET regulier zijn?

Is elke eindige taal regulier?

Is elke oneindige reguliere taal aftelbaar?

Is het mogelijk om gegeven een reguliere taal de bijhorende reguliere expressie te construeren?

Als je een string s krijgt en een reguliere expressie E, kan je dan (gemakkelijk) bepalen of $s \in L_E$?

Kan je alle strings in L_E genereren als je E krijgt?

5 De subalgebra van reguliere talen

De verzameling van talen over een alfabet Σ noteren we met L_Σ. Ze vormt een algebra: de verzameling zelf is $\mathcal{P}(\Sigma^*) = L_\Sigma$.

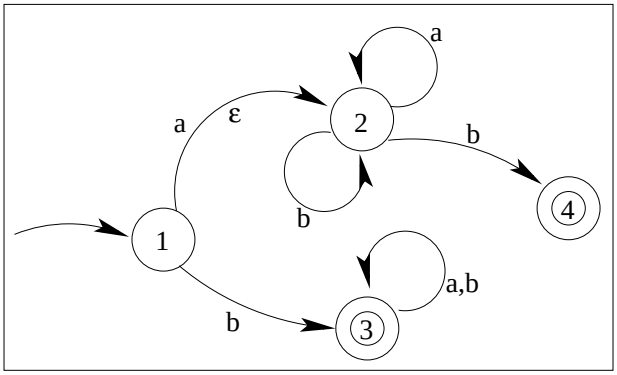
Als we twee talen L_1 en L_2 uit L_Σ nemen, dan kunnen we die gebruiken om de taal L_1L_2 te maken (concatenatie van talen), de taal $L_1 \cup L_2$ (unie van talen), de taal L_1^* (willekeurig lange concatenatie) en de complementstaal $\overline{L_1}$. Het resultaat zit terug in L_Σ. Dus: L_Σ is een algebra met (minstens) vier inwendige operaties.

Vermits $RegLan \subseteq L_\Sigma$ is het zinvol om te vragen of RegLan een subalgebra is van L_Σ: daarvoor moeten de operaties ook inwendig zijn op RegLan.

Formuleer de stelling die uitdrukt dat RegLan een subalgebra is van L_Σ en bewijs je stelling op een constructieve manier, t.t.z. construeer o.a. een E zodanig dat $L_E = L_{E_1} \cup L_{E_2}$. Heb je een probleem met het complement van een reguliere taal?

6 Eindige toestandsautomaten

Eindige toestandsautomaten zijn bedoeld om talen mee te beschrijven - testen en genereren van strings horen daarbij. Eindige toestandsautomaten kunnen grafisch voorgesteld worden en daarmee beginnen we: later geven we een formele definitie. Figuur 2.1 toont een eerste voorbeeld van een NFA[3] over het alfabet $\{a, b, c\}$.



Figuur 2.1: Een eindige toestandsautomaat

De belangrijke kenmerken:

- we zien een gerichte graaf
- de knopen hebben een naam (hier een getal) - de knopen noemen we toestanden
- er zijn twee soorten knopen: knopen met een dubbel cirkeltje getekend noemen we (aanvaardende) eindtoestanden
- lussen zijn toegestaan (bogen van een knoop naar dezelfde knoop)
- de bogen dragen een label (soms meer dan één); dat label is een symbool uit het alfabet, of meerdere symbolen uit het alfabet (gescheiden door een komma) en/of ϵ
- er is slechts één boog die niet vertrekt in een knoop; die boog komt aan in een knoop die we de starttoestand noemen

We gebruiken de grafische voorstelling van de NFA als volgt:

[3] Onze afkorting voor een eindige toestandsautomaat: we verklaren die later wel.

1. je krijgt een string s over het alfabet in handen en vertrekt ermee in de starttoestand
2. je mag nu van één toestand naar een andere gaan door een boog te volgen en in je vertrektoestand een symbool achter te laten dat op de boog staat en van voor in je string staat: je string wordt daardoor korter; als de boog ook ϵ bevat, dan hoef je niet een teken achter te laten
3. blijf overgangen maken: als je aankomt in een eindtoestand en je string is leeg op dat ogenblik, dan zeggen we *de NFA heeft de initiële string s aanvaard*

Dit geeft ons slechts een informele definitie van aanvaarde string!
Hier is een tweede manier om die grafische voorstelling te gebruiken:

1. vertrek met een lege string in de starttoestand
2. volg nu willekeurig bogen van de toestand waarin je bent, naar een volgende toestand: als op die boog een symbool staat, voeg het vanachter toe aan je huidige string; blijf rondlopen
3. telkens je in een eindtoestand arriveert, en je hebt string s opgebouwd ondertussen, roep luid *deze machine aanvaardt s*

Zelf doen: beantwoord

wordt ac door de NFA in figuur 2.1 aanvaard?

wordt bbb door de NFA in figuur 2.1 aanvaard?

zijn er verschillende manieren om bbb te aanvaarden?

kan het zijn dat je vast komt te zitten en wat zegt dat over bbb?

kan je strings geven die niet door de machine worden aanvaard?

maak een NFA die een kring bevat: kan je in een lus komen? is dat erg?

Informele definitie 6.1 De taal door een NFA M bepaald
Een taal L wordt bepaald door een NFA M, indien M elke string van L aanvaardt en geen andere strings. We noteren L_M.

Het is niet zo belangrijk dat de alfabetten van de NFA en de taal dezelfde zijn, maar we zullen het voor het gemak wel dikwijls veronderstellen.

Definitie 6.2 Equivalentie van twee NFA's
Twee NFA's worden **equivalent** genoemd als ze dezelfde taal bepalen

De notie *equivalentie van NFA's* bepaalt een equivalentierelatie op de NFA's en we kunnen de equivalentieklassen van de NFA's beschouwen onder die equivalentierelatie: elke equivalentieklasse komt nu overeen met één taal.

Zelf doen: denk na over

er bestaat een procedure om na te gaan of twee gegeven NFA's equivalent zijn

voor elke NFA bestaat een equivalente met hoogstens één eindtoestand

voor elke NFA bestaat een equivalente waarin je nooit *vast kan komen te zitten*

We hebben regelmatig de verzameling $\Sigma \cup \{\epsilon\}$ nodig: we zullen die afkorten door Σ_ϵ.

Definitie 6.3 Niet-deterministische eindige toestandsautomaat
Een **niet-deterministische eindige toestandsautomaat** is een 5-tal $(Q, \Sigma, \delta, q_s, F)$ waarbij

- Q een eindige verzameling toestanden is
- Σ is een eindig alfabet
- δ is de overgangsfunctie van de automaat, t.t.z. $\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$
- q_s is de starttoestand en natuurlijk een element van Q
- $F \subseteq Q$: F is de verzameling eindtoestanden

NFA is de afkorting van *Non-deterministic Finite Automaton*.

We definiëren nu ook formeel wat het betekent dat een string s wordt aanvaard door een NFA.

Definitie 6.4 Een string s wordt aanvaard door een NFA
Een string s wordt aanvaard door een NFA $(Q, \Sigma, \delta, q_s, F)$ indien s kan geschreven worden als $a_1a_2a_3...a_n$ met $a_i \in \Sigma_\epsilon$, en er een rij toestanden $t_1t_2t_2t_3...t_{n+1}$ bestaat zodanig dat

- $t_1 = q_s$
- $t_{i+1} \in \delta(t_i, a_i)$
- $t_{n+1} \in F$

Zelf doen:

Je hebt nu een intuïtieve notie van NFA's d.m.v. hun grafische voorstelling, en je hebt nu een formele definitie; zorg dat je intuïtie in overeenstemming is met de definitie. Doe hetzelfde met de notie van aanvaarde string.

Hierboven worden niet-deterministische automaten gedefinieerd: het is mogelijk dat in sommige toestanden je door een bepaald symbool achter te laten de keuze hebt tussen meerdere bogen, en/of dat je zelfs niks moet achterlaten. Sommige automaten die onder de definitie vallen, zijn echter deterministisch: je hebt in geen enkele toestand de keuze, t.t.z. het eerste symbool van je huidige string bepaalt altijd je volgende overgang (als er al één mogelijk is). Zulk een deterministische automaat korten we af door DFA.

Het vervolg van deze sectie brengt de notie van reguliere expressie en NFA samen: eerst construeren we vanuit een RE een NFA zodanig dat
$L_{RE} = L_{NFA}$. Daarna doen we het omgekeerde. Te samen bewijst dat dat de twee formalismen equivalent zijn.

7 De transitietabel

De δ van een NFA is een functie met een eindig domein, en kan gemakkelijk voorgesteld worden in tabelvorm: we noemen die tabel de transitietabel, omdat die aangeeft welke de overgangen zijn in de NFA . Een voorbeeld:

Q	Σ_ϵ	$\mathcal{P}(Q)$
1	a	$\{2\}$
1	b	$\{3\}$
1	ϵ	$\{2\}$
2	a	$\{2\}$
2	b	$\{2,4\}$
3	a	$\{3\}$
3	b	$\{3\}$
2	ϵ	$\emptyset$
3	ϵ	$\emptyset$
4	a	$\emptyset$
4	b	$\emptyset$
4	ϵ	$\emptyset$
1,2,3,4	c	$\emptyset$

Tabel 2.1: De transitietabel voor de NFA in Figuur 2.1

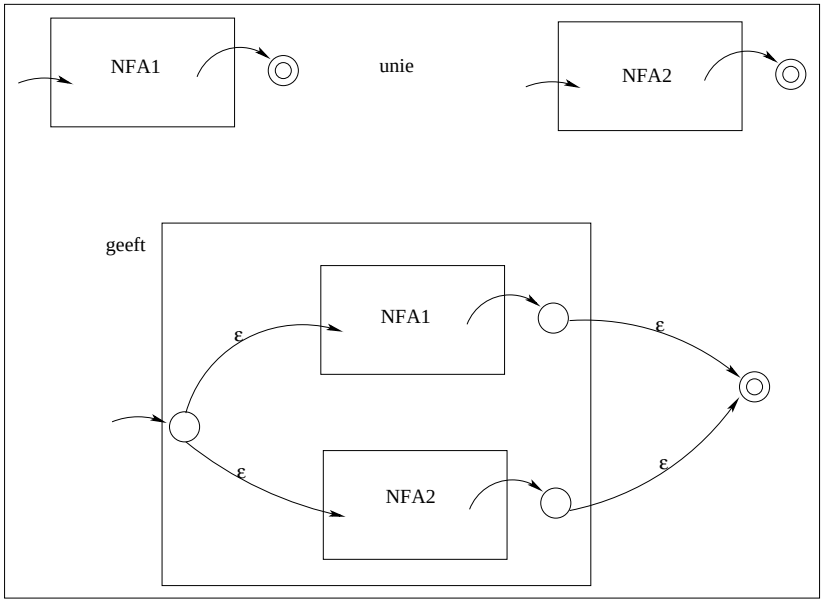
Voor elke toestand van de NFA in combinatie met een symbool uit Σ_ϵ waarvoor een boog bestaat in de grafische voorstelling, hebben we een overeenkomstige verzameling toestanden waarnaar de overgang mogelijk is. Als er geen boog is, dan kunnen we een entry in de tabel toevoegen met een lege verzameling van toestanden.

De transitietabel ziet eruit als iets dat we zouden kunnen gebruiken in een programma dat een NFA implementeert. Maar het niet-determinisme is nog storend: niet panikeren, we werken dat later wel weg.

8 De algebra van NFA's

Laat ons een vast alfabet kiezen: later kunnen we die afspraak eventueel wat afzwakken. De verzameling NFA's over dat alfabet is goed gedefinieerd: gebruik de definitie van NFA op pagina 14. We laten zien dat er op die verzameling drie inwendige operaties bestaan, die we *unie*, *concatenatie* en *ster* noemen. Ondertussen weten jullie dat een NFA altijd genoeg heeft aan één eindtoestand, waaruit bovendien geen pijlen vertrekken. Dat maakt het iets gemakkelijker.

De unie van twee NFA's: Figuur 2.2 laat de intuïtie zien achter hoe de unie van twee NFA's kan genomen worden: maak één nieuwe eindtoestand en teken een ϵ-boog tussen de oude eindtoestanden en de nieuwe. Maak van de oude eindtoestanden gewone toestanden. Maak een nieuwe begintoestand en verbind die met een ϵ-boog met de oude begintoestanden (die worden daardoor gedegradeerd naar gewone toestanden).



Figuur 2.2: Unie van twee NFA's

Formeel schrijven we:
Gegeven $NFA_1 = (Q_1, \Sigma, \delta_1, q_{s1}, \{q_{f1}\})$ en $NFA_2 = (Q_2, \Sigma, \delta_2, q_{s2}, \{q_{f2}\})$.
De unie $NFA_1 \cup NFA_2$ is de $NFA = (Q, \Sigma, \delta, q_s, F)$ waarbij

- $Q = Q_1 \cup Q_2 \cup \{q_s, q_f\}$ waarbij q_s en q_f nieuwe toestanden zijn

- $F = \{q_f\}$
- δ is gedefinieerd als:
  $\delta(q, x) = \delta_i(q, x) \quad \forall q \in Q_i \backslash \{q_{fi}\}, x \in \Sigma_\epsilon$ voor i=1,2

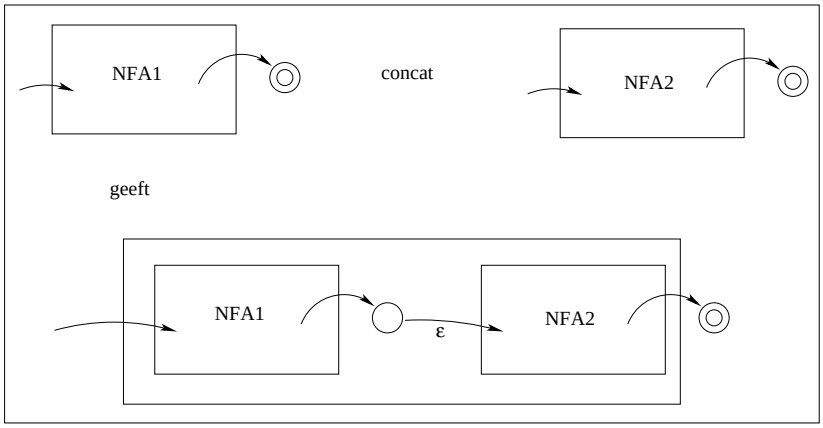
  $\delta(q_s, \epsilon) = \{q_{s1}, q_{s2}\}$

  $\delta(q_s, x) = \emptyset \quad \forall x \in \Sigma$

  $\delta(q_{fi}, \epsilon) = \{q_f\}$ voor i = 1,2

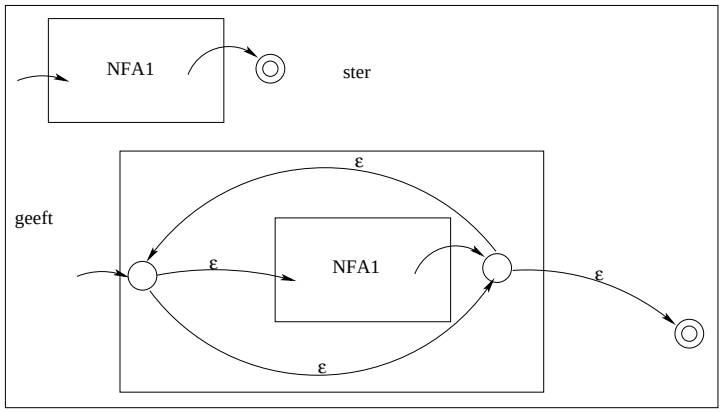
  $\delta(q_{fi}, x) = \emptyset \quad \forall x \in \Sigma$ en voor i = 1,2

De concatenatie van twee NFA's: Deze keer geven we alleen de visuele representatie van de concatenatie in figuur 2.3:



Figuur 2.3: Concatenatie van twee NFA's

De ster van een NFA: weerom geven we enkel de visuele representatie, in figuur 2.4.



Figuur 2.4: De ster van een NFA

Werk de formele beschrijvingen van concatenatie en de ster zelf uit.

Zelf doen:

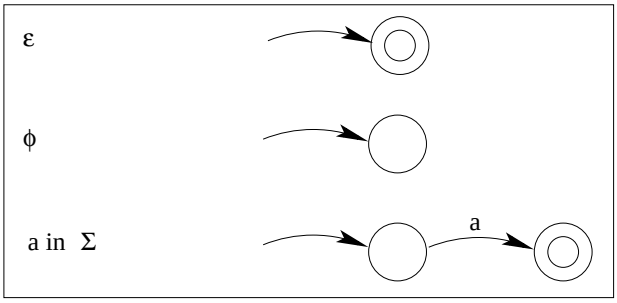
De concatenatie van NFA_1 en NFA_2 bepaalt $L_{NFA_1}L_{NFA_2}$. Bewijs dat.

Formuleer iets analoogs voor de ster en de unie.

Wat bewijst dat over de algebraïsche isomorfie tussen ... en ...?

9 Van reguliere expressie naar NFA

We hebben alle ingrediënten om van een reguliere expressie RE een NFA te maken, en zodanig dat de $L_{RE} = L_{NFA}$. Vermits reguliere expressies inductief gedefinieerd zijn (zie definitie pagina 8) zullen we voor elk lijntje van die definitie een overeenkomstige NFA definiëren. We gebruiken de notatie NFA_{RE} om de NFA aan te duiden die overeenkomt met de reguliere expressie RE.
Figuur 2.5 geeft de NFA voor de eerste drie basisgevallen in de definitie op pagina 8:



Figuur 2.5: Een NFA voor de drie basisgevallen

De drie recursieve gevallen beschrijven we als volgt: laat E_1 en E_2 twee reguliere expressies zijn, dan is

- $NFA_{E_1E_2} = concat(NFA_{E_1}, NFA_{E_2})$
- $NFA_{E_1^*} = ster(NFA_{E_1})$
- $NFA_{E_1|E_2} = unie(NFA_{E_1}, NFA_{E_2})$

Stelling 9.1
De constructie hierboven bewaart de taal, t.t.z.

$$L_{NFA_E} = L_E.$$

Bewijs Geef zelf een bewijs door structurele inductie. ■

10 Van NFA naar reguliere expressie

De weg omgekeerd is iets meer complex: we voeren eerst een nieuwe soort van eindige automaten in - de GNFA's, de G staat voor gegeneraliseerd. Daarna zullen we het volgende traject doorlopen:

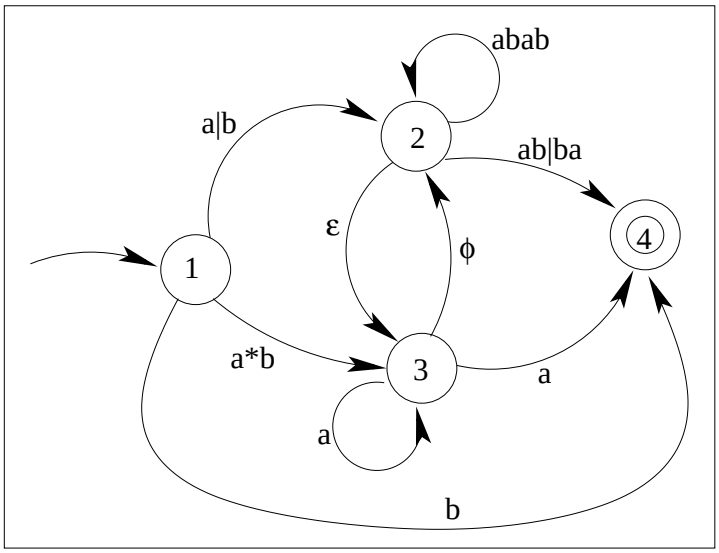
$NFA \rightarrow GNFA \rightarrow GNFA\ met\ 2\ toestanden \rightarrow reguliere\ expressie$

In elk van die stappen zullen we hard moeten maken dat de taal beschreven door het formalisme niet verandert.

Informele definitie 10.1 GNFA
Een **GNFA** is een eindige toestandsmachine met de volgende wijzigingen en beperkingen:

- er is slechts één eindtoestand en die is verschillend van de starttoestand
- er is juist één boog van de starttoestand naar elke andere toestand, maar er komen geen pijlen aan (behalve de startpijl)
- er is juist één boog van elke toestand naar de eindtoestand maar uit de eindtoestand vertrekken geen pijlen
- tussen elke andere twee toestanden is er juist één boog in beide richtingen
- er is ook juist één boog van elke andere toestand naar zichzelf
- de bogen hebben als label een reguliere expressie

Figuur 2.6 toont een GNFA.



Figuur 2.6: Een GNFA

We gebruiken de (grafische voorstelling van de) GNFA als volgt:

1. je krijgt een string s over het alfabet en vertrekt ermee in de starttoestand
2. je mag nu van één toestand naar een andere overgaan door een boog te volgen en in je vertrektoestand een rij symbolen die van voor op je string voorkomen achter te laten; die rij symbolen moet voldoen aan de reguliere expressie die op de boog staat; je string wordt daardoor korter; als de boog ook ϵ bevat, dan hoef je niet een teken achter te laten; als de boog enkel maar ϕ bevat, dan kan je de boog niet nemen
3. blijf overgangen maken: als je aankomt in de eindtoestand en je string is leeg op dat ogenblik, dan zeggen we *de GNFA heeft de initiële string s aanvaard*

Zelf doen:

Zoek voor de GNFA in figuur 2.6 strings die aanvaard worden en strings die niet aanvaard worden.

We beschrijven nu een algoritme om van een gegeven NFA een RE te maken:

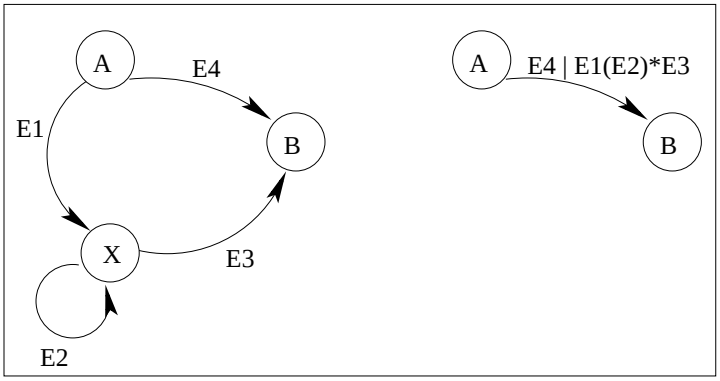
Stap 1: *Maak van de NFA een GNFA*

Voer een nieuwe starttoestand in en een nieuwe (unieke) eindtoestand. Teken een ϵ-boog van de nieuwe begintoestand naar de oude begintoestand, en van elke oude eindtoestand naar de nieuwe eindtoestand. Teken de ontbrekende bogen met een ϕ. Als er nu tussen twee toestanden twee of meer parallelle gerichte bogen zijn, neem die dan samen met als label de unie van de labels van de parallelle bogen.

Stap 2: *Reduceer de GNFA*

Kies een willekeurige toestand X verschillend van de start- of eindtoestand - als er geen meer zijn, ga naar stap 3. Verwijder die knoop als volgt: kies toestanden A en B zodat er bogen zijn van A naar B met label E_4, van A naar X met E_1, van X naar zichzelf met E_2 en van X naar B met E_3. Vervang het label op de boog van A naar B door $E_4 \mid E_1 E_2^* E_3$. Doe dit voor alle koppels A en B. Verwijder daarna de knoop X met alle bogen die erin toekomen of vertrekken.

De basisstap wordt geïllustreerd in figuur 2.7.



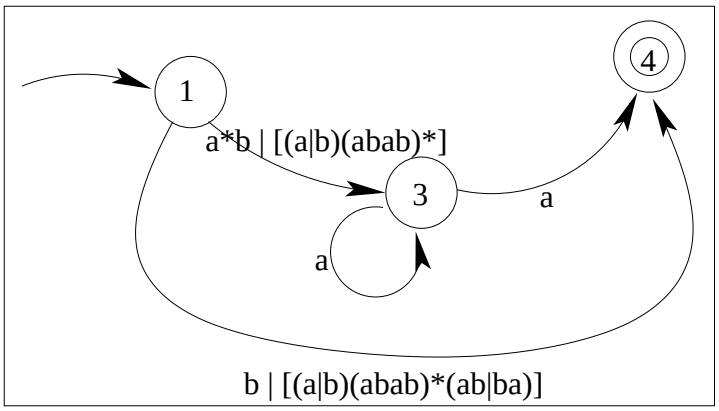
Figuur 2.7: Verwijdering van één toestand uit een GNFA

Herhaal stap 2.

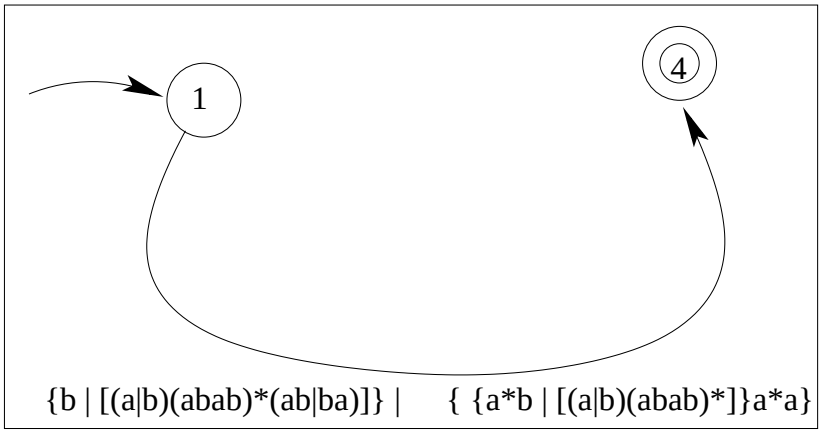
Stap 3: *Bepaal RE*

De GNFA heeft nu juist 2 toestanden (start- en eindtoestand) en daartussen één boog; die boog heeft een RE als label; dit is de RE die we zochten.

Voorbeeld: We passen dit stapsgewijze toe op de GNFA van figuur 2.6: figuren 2.8 en 2.9 tonen dat.



Figuur 2.8: Toestand 2 is verwijderd



Figuur 2.9: Toestand 3 is verwijderd

We moeten nog bewijzen dat de reductie met één toestand in Stap 2 in het algoritme de verzameling aanvaarde strings niet verandert. We moeten daarvoor twee dingen bewijzen: (1) indien een string s aanvaard werd voor de reductie, dan ook na de reductie; (2) indien een string s niet aanvaard werd voor de reductie, dan ook niet na de reductie.

We gebruiken de notie van het pad doorheen de toestanden dat je kan volgen om een string te accepteren (die notie staat voor een NFA in de definitie op pagina 15 - schrijf ze hier eens uit voor een GNFA): zulk een acceptatiepad is dus een opeenvolging van toestanden. We verwijzen naar de machine voor de reductie met $GNFA_{voor}$ en naar de machine erna met $GNFA_{na}$.

1. Als s aanvaard werd door $GNFA_{voor}$ met een pad dat X niet bevat, dan wordt s door datzelfde pad in $GNFA_{na}$ aanvaard.

   Als dat pad X bevat, dan zijn er toestanden A en B, zodat AX^{n}B ($n > 0$) een opeenvolging is in het pad. De reguliere expressies op de bogen AX, XX, XB zijn E1, E2, E3 en bijgevolg kost van A naar B gaan langs X een stukje string dat voldoet aan $E1(E2)^*E3$: die reguliere expressie staat ook in de boog AB in $GNFA_{na}$, dus ...

2. Als s aanvaard wordt door $GNFA_{na}$ dan bevat het acceptatiepad uiteraard alleen maar toestanden verschillend van X. Op een boog van A naar B (twee opeenvolgende toestanden in het acceptatiepad) staat de reguliere expressie $E4|E1(E2)^*E3$: die gebruiken betekent een stukje string uitgeven dat voldoet aan E4 of aan $E1(E2)^*E3$, dus in $GNFA_{voor}$ komt dat overeen met ofwel de boog AB volgen ofwel de bogen AX, XX (zo dikwijls als nodig) en XB. AB had als label $E4$, AX heeft E1, ...

   Dus als een string aanvaard wordt door $GNFA_{na}$ dan ook door $GNFA_{voor}$.

We moeten ook nog stap 1 verantwoorden, t.t.z. argumenteren dat door de NFA om te vormen naar een GNFA, de taal niet verandert. Doe dat zelf.

Besluit: twee formalismen (NFA en RE) bepalen precies dezelfde klasse van talen die we de reguliere talen hebben genoemd. We hebben dat bewezen en de bewijzen zijn constructief: we kunnen de bewijzen gemakkelijk omvormen tot programma's in Java, Prolog ... om vanuit een RE een NFA te berekenen, of omgekeerd. We kunnen echter beter doen dan tot nu toe: onze NFA's zijn niet-deterministisch en het zou leuk zijn als we genoeg hebben aan deterministische automaten. In sectie 11 zullen we dit uitspitten. Tenslotte kunnen we ook een minimaliteitscriterium voor deterministische automaten bestuderen: dat gebeurt in secties 12 en 13.

11 Deterministische eindige toestandsmachines

De definitie van een eindige toestandsmachine NFA laat toe dat vanuit een bepaalde toestand bogen vertrekken met ϵ en ook meer dan één boog met (bijvoorbeeld) het label a (a in het alfabet). Dat is de bron van niet-determinisme: als in die toestand het eerste symbool van je huidige string a is, dan heb je de keuze: a achterlaten en nog keuze tussen welke boog met a erop volgen, of niets achterlaten en de ϵ-boog volgen. Dit implementeren (bijvoorbeeld op basis van de transitietabel) is niet moeilijk, maar als je alle mogelijkheden wil uitproberen, dan moet je op je stappen kunnen terugkeren. Ook weer niet onoverkomelijk, maar het is duidelijk dat dit niet tot optimale programma's zal leiden. Het ware efficiënter als er in elke toestand slechts één mogelijkheid bestond per symbool en geen ϵ -overgangen. We beperken dus de klasse van automaten tot de deterministische eindige toestandsmachines - we noteren DFA - door geen ϵ-overgangen toe te laten en bovendien mag een symbool van het alfabet hoogstens op één uitgaande boog per toestand staan. Formeel doen we dat door te schrijven dat $\delta : Q \times \Sigma \to Q$ een partiële functie is. Het zou moeten duidelijk zijn dat een taal bepaald door een DFA ook regulier is. De volgende vraag is daarmee nog niet beantwoord: kan elke reguliere taal bepaald worden door een DFA?

Een andere manier om daar tegenaan te kijken is de volgende: gegeven een reguliere taal L, dan bestaat er een reguliere expressie E voor L; voor die E kunnen we gemakkelijk de automaat NFA_E maken (zie pagina 20). Laat ons nu proberen die NFA_E om te vormen tot een DFA die dezelfde taal (L) aanvaardt. Als dat lukt hebben we bewezen dat elke reguliere taal door een DFA bepaald wordt.

We beschrijven de transformatie van een NFA naar de DFA in het algemeen.

Gegeven: een NFA $= (Q_n, \Sigma, \delta_n, q_{sn}, F_n)$

Gevraagd: een DFA $= (Q_d, \Sigma, \delta_d, q_{sd}, F_d)$ zodanig dat $L_{NFA} = L_{DFA}$

Constructie: $Q_d = \mathcal{P}(Q_n)$: elke toestand in de DFA is een verzameling toestanden van de NFA

$F_d = \{S | S \in Q_d, S \cap F_n \neq \emptyset\}$: een eindtoestand in de DFA bevat altijd een eindtoestand van de NFA

Dat laat ons nog δ_d. Voor de duidelijkheid: $\delta_d : (\mathcal{P}(Q_n) \times \Sigma) \to \mathcal{P}(Q_n)$

We voeren eerst een afbeelding $eb : Q_n \to \mathcal{P}(Q_n)$ in (eb staat voor **e**psilon-**b**ereikbaar):

- $eb(q)$ is de verzameling toestanden in NFA die met nul, één of meer ϵ-bogen bereikbaar zijn vanuit q
- We liften de definitie van eb naar $\mathcal{P}(Q_n)$ op de gewone manier: voor een $\mathcal{Q} \in \mathcal{P}(Q_n)$
$$eb(\mathcal{Q}) = \cup_{q \in \mathcal{Q}} eb(q)$$
- δ_n liften we op dezelfde manier.

Vervolgens definiëren we δ_d als volgt:

- $\delta_d(\mathcal{Q}, a) = eb(\delta_n(\mathcal{Q}, a))$ [4] voor $\mathcal{Q} \in Q_d$
- in woorden: vanuit een toestand $\mathcal{Q}$ in de DFA ga je naar een volgende toestand in de DFA door voor elke NFA toestand in $\mathcal{Q}$ eerst de overgangsfunctie van de NFA te gebruiken, en daarna de ϵ-bogen te volgen - van al die resulterende toestandsverzamelingen neem je de unie.

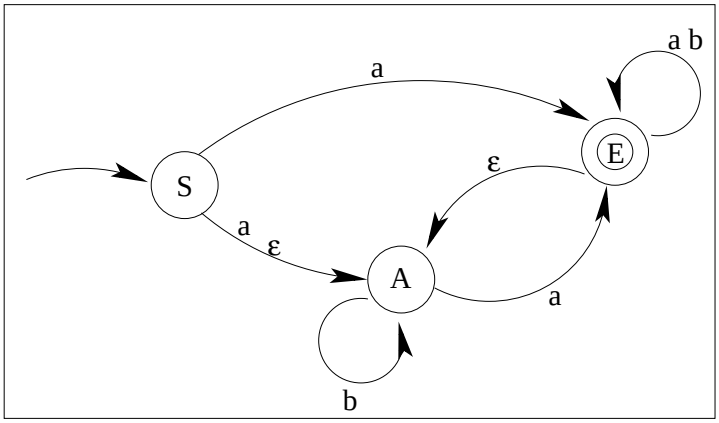
Tenslotte definiëren we

$$q_{sd} = eb(q_{sn}).$$

Einde

[4] Kijk de signatuur na!

We passen die constructie toe op de NFA in figuur 2.10. Die NFA heeft 3 toestanden, dus hebben we in principe 8 toestanden in de resulterende DFA.



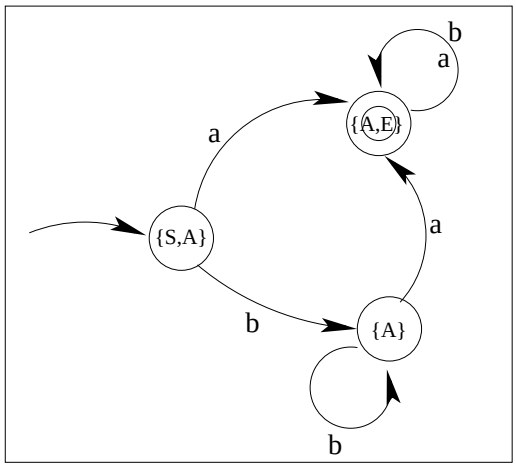
Figuur 2.10: Een NFA

Tabel 2.2 laat het resultaat zien van de constructie van de verschillende onderdelen van de DFA.

$q \in Q_d$	$eb(q)$	$\delta_n(q,a)$	$\delta_n(q,b)$	$\delta_d(q,a)$	$\delta_d(q,b)$
{}	{}	{}	{}	{}	{}
{S}	{S,A}	{A,E}	{}	{A,E}	{}
{A}	{A}	{E}	{A}	{A,E}	{A}
{E}	{A,E}	{E}	{E}	{A,E}	{A,E}
{S,A}	{S,A}	{A,E}	{A}	{A,E}	{A}
{S,E}	{S,A,E}	{A,E}	{E}	{A,E}	{A,E}
{A,E}	{A,E}	{E}	{A,E}	{A,E}	{A,E}
{S,A,E}	{S,A,E}	{A,E}	{A,E}	{A,E}	{A,E}

Tabel 2.2: De onderdelen van de DFA voor de NFA in Figuur 2.10

Een aantal toestanden is niet bereikbaar vanuit de starttoestand $\{S, A\}$. Het is voldoende om enkel de bereikbare toestanden voor te stellen: de grafische voorstelling van de DFA is te zien in figuur 2.11.



Figuur 2.11: De resulterende DFA

Zelf doen:

Bepaal welke taal deze automaat beschrijft. Of maak er een reguliere expressie van en druk dan in woorden uit welke strings aanvaard worden. Is dit de *kleinste* automaat die deze taal bepaalt? Wat is volgens jou een goede notie van *kleinste* automaat?

De constructie vereist maximaal $2^{\#Q_n}$ toestanden in de DFA, maar het voorbeeld toont dat het voorkomt dat Q_d niet veel groter hoeft te zijn dan Q_n, als we de niet bereikbare toestanden niet opnemen. Is het mogelijk dat de DFA *minder* toestanden heeft dan de NFA waarvan we vertrokken?

Uitbreiding van δ naar strings: voor een DFA heeft δ als domein $Q \times \Sigma$. Het is handig δ uit te breiden tot een functie δ^* op het domein $Q \times \Sigma^*$ als volgt:

- $\delta^*(q, \epsilon) = q$
- $\delta^*(q, aw) = \delta^*(\delta(q, a), w)$ indien $\delta(q, a)$ bestaat - hierin is $a \in \Sigma$ en $w \in \Sigma^*$.

Zelf doen:

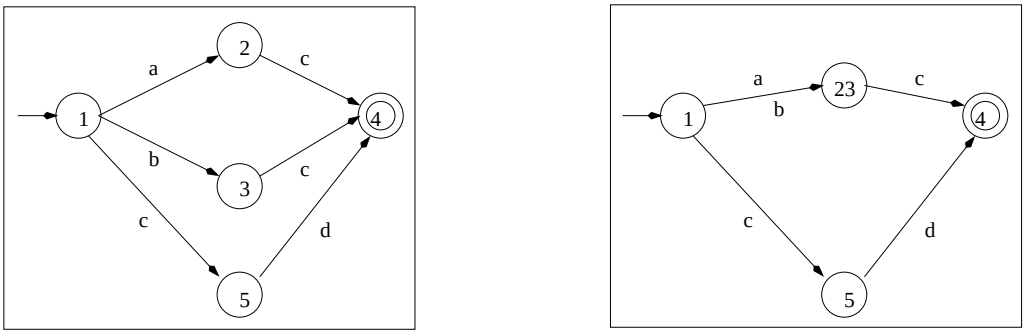
Bewijs dat $\delta^*(q, wa) = \delta(\delta^*(q, w), a)$ voor $a \in \Sigma$ en $w \in \Sigma^*_\epsilon$

12 Minimale DFA

Voor een gegeven reguliere taal L bestaan er meestal veel DFA's die de taal bepalen[5]. Het is belangrijk om kleine machines te maken: als je in een toepassing een DFA nodig hebt, dan moet je op een of andere manier de toestanden voorstellen en dus heb je er belang bij het aantal toestanden laag te houden. Het is zonder meer duidelijk dat er voor een gegeven reguliere taal een DFA bestaat met het minimale aantal toestanden. De vraag is hoe we een minimale DFA construeren, en bewijzen dat de constructie een minimale DFA oplevert. We proberen minimaliteit te verkrijgen door toestanden weg te halen uit de machine.

Toestanden die niet bereikbaar zijn vanuit q_s zijn nutteloos: die toestanden kunnen we zonder meer wegdoen.

Om nog meer toestanden weg te doen, eerst een voorbeeld: Figuur 2.12 toont links een DFA met 5 toestanden.



Figuur 2.12: Links een DFA met 2 equivalente toestanden; rechts zijn ze samengenomen

Vanuit de toestanden 2 en 3 vertrekken bogen met hetzelfde label naar de eindtoestand. Dat geeft aan dat die twee toestanden *f-gelijk* zijn, t.t.z. eens je in 2 of 3 geraakt bent, geraak je met dezelfde strings tot aan de eindtoestand: de f in f-gelijk staat voor *finaal*. Langs de andere kant: om van 5 naar de eindtoestand te gaan heb je een andere string nodig dan om van 3 naar de eindtoestand te gaan, en we noemen 3 en 5 dan ook *f-verschillend*.[6] De idee van minimalisatie van een DFA is nu: identificeer verzamelingen van f-gelijke toestanden en neem

[5] Soms oneindig veel?

[6] In de literatuur vind je indistinguishable en distinguishable.

die samen.

Voor het gemak zullen we eisen dat vanuit elke toestand er een boog vertrekt voor elk symbool van het alfabet, m.a.w. dat δ een totale functie is; overtuig je ervan dat je hoogstens één extra toestand nodig hebt om een DFA die die eigenschap niet heeft naar een DFA om te vormen met die eigenschap[7].

Laat ons eerst exact definiëren wanneer twee toestanden f-verschillend zijn en wanneer f-gelijk:

Definitie 12.1 f-verschillende en f-gelijke toestanden
Twee toestanden p en q zijn **f-gelijk** indien
$$\forall w \in \Sigma^* : \delta^*(p, w) \in F \iff \delta^*(q, w) \in F$$
Twee toestanden zijn **f-verschillend** indien ze niet f-gelijk zijn.

Als p en q f-verschillend zijn, dan wil dat zeggen dat er een woord w bestaat zodanig dat
$$\delta^*(p, w) \in F \quad en \quad \delta^*(q, w) \notin F \text{ of omgekeerd.}$$

Nu in woorden een algoritme om sets van f-gelijke toestanden te vinden:

Init: een toestand p die geen eindtoestand is, is zeker f-verschillend van elke eindtoestand; alle andere paren toestanden zijn nog onbeslist

Repeat: neem een paar toestanden p en q dat nog onbeslist is: stel dat er een symbool a bestaat zodanig dat je met dat symbool van p en q gaat naar twee f-verschillende toestanden, dan beslis je dat p en q f-verschillend zijn

Consolideer: voor elk paar toestanden p en q waarvoor je nog niet beslist had, beslis nu dat p en q f-gelijk zijn; gebruik die gelijkheidsrelatie om Q (de toestanden) te partitioneren, t.t.z. te verdelen in disjuncte Q_i die samen Q uitmaken; de Q_i vormen de toestanden van de minimale DFA; hoe δ eruit ziet komt later meer formeel

Om wat schrijfwerk uit te sparen zullen we de notatie p_a gebruiken als afkorting voor $\delta(p, a)$.

[7] Veel auteurs beschouwen van in het begin enkel DFA's met die eigenschap en relaxen die later voor het gemak.

Algoritme 12.2 F-gelijke toestanden

Init: Beschouw de graaf V met als knopen de toestanden van de DFA; voeg een boog toe tussen elke twee knopen waarvan er precies één in F zit; label die boog met ϵ ; een boog tussen knopen x en y, met een label l, zullen we aanduiden door (x, y, l)

Repeat: Indien er knopen p en q zijn waarvoor geldt dat

- er is geen boog tussen p en q
- $\exists a \in \Sigma : \exists(p_a, q_a, \_) \in V$

kies dan een a waarvoor geldt dat $(p_a, q_a, \_) \in V$ en voeg de boog (p, q, a) toe aan V; ga terug naar het begin van Repeat;

anders: ga naar Gelijk;

Gelijk: Beschouw de graaf G die als knopen heeft de toestanden uit Q en die een boog heeft tussen twee knopen p en q indien V **geen** boog heeft tussen p en q (m.a.w. de complementsgraaf); elke component van G is een klik (een volledig verbonden graaf, isomorf met K_n voor een n); laat Q_i de verzameling knopen in component i voorstellen; alle toestanden binnen één Q_i zijn f-gelijk; elke toestand in Q_i is f-verschillend van elke toestand in Q_j voor $i \neq j$

Bewijs De eindigheid van het algoritme is evident: in Repeat wordt één boog toegevoegd en het maximaal aantal bogen dat V kan hebben is $N(N-1)/2$ met N het aantal knopen van V.

We bewijzen dat

$$(p, q, \_) \text{ is een boog in } V \iff p \text{ en } q \text{ zijn f-verschillend}$$

$\Longrightarrow$: indien (p, q, X) een boog is in V, dan zijn er twee mogelijkheden: $X = \epsilon$ of $X = a \in \Sigma$; in het eerste geval hebben we onmiddellijk dat p en q f-verschillend zijn (neem daarvoor $w = \epsilon$ in het besluit na de definitie van f-gelijk op pagina 31); in het tweede geval hebben we dat er een boog bestaat van de vorm $(p_a, q_a, \_)$ in V: je ziet nu dat je het label kunt gebruiken om in het vorige basisgeval terecht te komen, dus $\exists w \in \Sigma^* : \exists(p_w, q_w, \epsilon) \in V$, dus p en q zijn f-verschillend.

$\Longleftarrow$: indien p en q f-verschillend zijn, dan bestaat er een w zodat $\delta^*(p, w) \in F \quad en \quad \delta^*(q, w) \notin F$ of omgekeerd; als die w de lege string is, dan is het besluit onmiddellijk; in het andere geval heeft w een laatste symbool $z \in \Sigma$ en

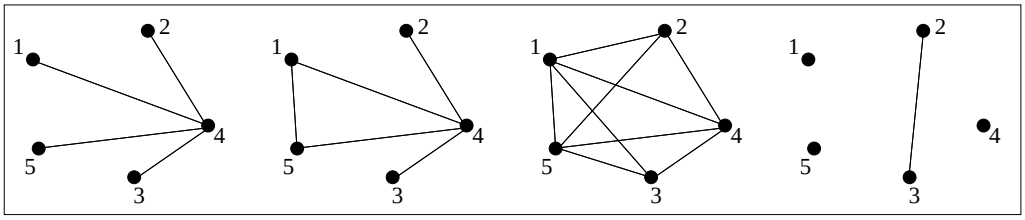
kan geschreven worden als $w = vz$; we hebben dan $\delta^*(p, w) = \delta(\delta^*(p, v), z)$ dus: tussen de toestanden $\delta^*(p, v)$ en $\delta^*(q, v)$ is er een boog; we kunnen nu dat laatste symbool van v kwijtgeraken enzovoort tot we zullen uitkomen op: er is een boog tussen $\delta^*(p, \epsilon)$ en $\delta^*(q, \epsilon)$ en we zijn klaar. ■

We hebben nu in de DFA de equivalentieklassen van f-gelijke toestanden gevonden - de Q_i op het einde van het algoritme - en zijn nu klaar om de DFA_{min} te definiëren: we vertrekken van een DFA $(Q, \Sigma, \delta, q_s, F)$ zonder onbereikbare toestanden.

DFA_{min} bestaat uit $(\tilde{Q}, \Sigma, \tilde{\delta}, \tilde{q_s}, \tilde{F})$ waarbij

- $\tilde{Q} = \{Q_1, Q_2, ...\}$ waarbij de Q_i verkregen zijn in het algoritme
- $\tilde{\delta}(Q_i, a) = Q_j$ waarbij Q_j verkregen wordt door: neem een $q \in Q_i$ (*) en neem dan de Q_j zodanig dat $\delta(q, a) \in Q_j$
- $\tilde{q_s}$ is de Q_i waarvoor geldt dat $q_s \in Q_i$
- $\tilde{F}$ is de verzameling van Q_i waarvoor geldt dat $Q_i \cap F \neq \emptyset$

Figuur 2.13 illustreert voor de DFA van figuur 2.12 hoe V evolueert en hoe de complementsgraaf er uitziet.



Figuur 2.13: 4 tussenstappen in het algoritme

Zelf doen:

In (*) hierboven is het niet belangrijk welke $q \in Q_i$ gekozen wordt: bewijs.

Bewijs ook dat als $Q_i \cap F \neq \emptyset$, dan $Q_i \cap F = Q_i$ (m.a.w. elk element van Q_i zit in F).

Bewijs dat in DFA_{min} elke twee toestanden f-verschillend zijn.

Bewijs tenslotte dat $L_{DFA} = L_{DFA_{min}}$

Wat als je DFA ook onbereikbare toestanden had en je voert die minimisatieprocedure uit?

We zouden nu graag bewijzen dat de voorheen geconstrueerde DFA_{min} een minimaal aantal toestanden heeft. We bewijzen een iets algemenere stelling:

Stelling 12.3 Als $DFA_1 = (Q_1, \Sigma, \delta_1, q_s, F_1)$ een machine is zonder onbereikbare toestanden en waarin elke twee toestanden f-verschillend zijn, dan bestaat er geen machine met strikt minder toestanden die dezelfde taal bepaalt.

Bewijs Laat DFA_1 als toestanden hebben $\{q_s, q_1, ..., q_n\}$, waarbij q_s de starttoestand is. Stel dat $DFA_2 = (Q_2, \Sigma, \delta_2, p_s, F_2)$ minder toestanden heeft dan DFA_1.

Vermits in DFA_1 elke toestand bereikbaar is, bestaan er strings $s_i, i = 1..n$ zodanig dat $\delta_1^*(q_s, s_i) = q_i$.

Vermits DFA_2 minder toestanden heeft moet voor een $i \neq j$
$\delta_2^*(p_s, s_i) = \delta_2^*(p_s, s_j)$.

Vermits q_i en q_j f-verschillend zijn, bestaat een string v zodanig dat

$\delta_1^*(q_i, v) \in F_1 \wedge \delta_1^*(q_j, v) \notin F_1$ of omgekeerd.

Dus ook $\delta_1^*(q_s, s_i v) \in F_1 \wedge \delta_1^*(q_s, s_j v) \notin F_1$ of omgekeerd. Dit betekent dat DFA_1 van de strings $s_i v$ en $s_j v$ er juist één accepteert.

Maar: $\delta_2^*(p_s, s_i v) = \delta_2^*(\delta_2^*(p_s, s_i), v) = \delta_2^*(\delta_2^*(p_s, s_j), v) = \delta_2^*(p_s, s_j v)$ hetgeen betekent dat DFA_2 ofwel beide strings $s_i v$ en $s_j v$ accepteert, of beide verwerpt.

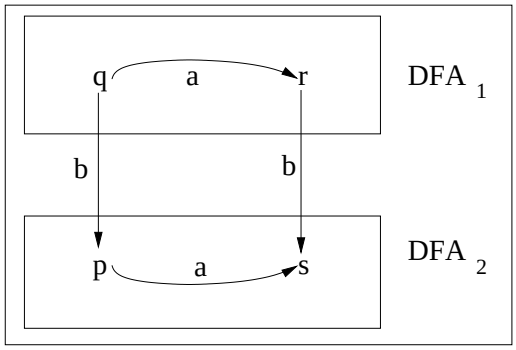
Dus kunnen DFA_1 en DFA_2 niet dezelfde taal bepalen. ■

De vroeger geconstrueerde DFA_{min} heeft geen onbereikbare toestanden en elke twee toestanden zijn f-verschillend, dus heeft DFA_{min} een minimaal aantal toestanden.

Twee DFA's zijn pas echt gelijk als hun toestanden dezelfde zijn, hun δ en hun finale toestanden. Maar toch kunnen twee DFA's zo hard op elkaar gelijken dat hun grafische voorstelling er hetzelfde uitziet op de naam van de toestanden na. Om dat uit te drukken is er de notie van isomorfe DFA's.

Definitie 12.4 Isomorfisme DFA's
$DFA_1 = (Q_1, \Sigma, \delta_1, q_{s1}, F_1)$ is **isomorf** met $DFA_2 = (Q_2, \Sigma, q_{s2}, \delta_2, F_2)$ indien er een bijectie $b : Q_1 \to Q_2$ bestaat zodanig dat

- $b(F_1) = F_2$
- $b(q_{s1}) = q_{s2}$
- $b(\delta_1(q, a)) = \delta_2(b(q), a)$ (zie figuur 2.14)



Figuur 2.14: Commutatief diagram voor b en δ_i

Het is gemakkelijk om in te zien dat twee isomorfe DFA's dezelfde taal bepalen.

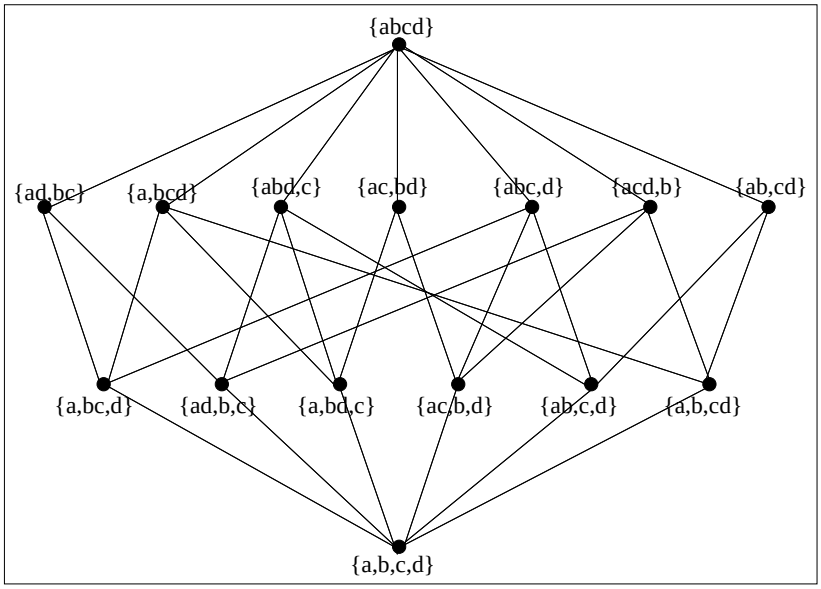
We kunnen nu rechtsreeks bewijzen dat de minimale DFA uniek is op isomorfisme na - probeer het! Maar het kan ook langs een elegante omweg. De bagage daarvoor krijg je in de volgende sectie.

13 Myhill-Nerode relaties op Σ^*

Deze topic vind je niet in elk boek over (reguliere) talen en in de boeken die het behandelen wordt het dikwijls beschouwd als een *advanced topic*. Het is goed om minstens één advanced topic te doorworstelen en we hebben voor de Myhill-Nerode relaties gekozen omdat het uiterst relevant is voor minimizatie van DFA's, en omdat we hier stukjes *oude* kennis kunnen hergebruiken: kennis van (equivalentie) relaties, (complete) tralies, ... Bovendien leert het ons veel over de structuur van de reguliere talen versus de DFA's. We duiken er direct in.

13.1 De partities van een verzameling vormen een complete tralie

Neem als voorbeeld de verzameling met 4 elementen $\{a, b, c, d\}$. Alle partities kan je opsommen:
$\{\{a, b, c, d\}\}$, $\{\{b, c, d\}, \{a\}\}$, $\{\{a, c, d\}, \{b\}\}$, $\{\{c, d\}, \{a, b\}\}$, $\{\{c, d\}, \{b\}, \{a\}\}$,
$\{\{a, b, d\}, \{c\}\}$, $\{\{b, d\}, \{a, c\}\}$, $\{\{b, d\}, \{c\}, \{a\}\}$, $\{\{a, d\}, \{b, c\}\}$,
$\{\{d\}, \{a, b, c\}\}$, $\{\{d\}, \{b, c\}, \{a\}\}$, $\{\{a, d\}, \{c\}, \{b\}\}$, $\{\{d\}, \{a, c\}, \{b\}\}$,
$\{\{d\}, \{c\}, \{a, b\}\}$, $\{\{d\}, \{c\}, \{b\}, \{a\}\}$
en je kan die in een Hasse diagram steken als in de figuur 2.15.



Figuur 2.15: Hasse diagram van de partities van $\{a, b, c, d\}$

Definitie 13.1 Een partitie P_1 is **fijner** dan P_2 indien elke element van P_1 vervat zit in een element van P_2

Zo is $\{a, b, c, d\}$ fijner dan elke andere partitie, en $\{ab, c, d\}$ is fijner dan $\{ab, cd\}$. In figuur 2.15 betekent fijner lager in het Hasse diagram.

Het omgekeerde van fijn is *grof* [8]. De grofste partitie bevat slechts één element en staat aan de top van het diagram.

Vermits er een 1-1 verband is tussen partities van een verzameling S en de equivalentierelaties op S, kunnen we ook zien dat de equivalentierelaties op S een complete tralie vormen met dezelfde structuur.

We nemen nu als verzameling Σ^* en bekijken daarop partities (of equivalentierelaties) die verband houden met een gegeven reguliere taal.

13.2 Partities van Σ^*

Σ^* is oneindig groot, dus zijn er oneindig veel partities van Σ^*; sommige partities tellen eindig veel elementen, andere partities zijn oneindig. Het kan helpen als je voor $\Sigma = \{a\}$ eens wat partities opschrijft voor Σ^*.

We vertrekken van een DFA $(Q, \Sigma, \delta, q_s, F)$ die een reguliere taal L accepteert. De eerste partitie van Σ^* die met L te maken heeft, is simpelweg $\{L, \overline{L}\}$. Deze partitie geeft aanleiding tot een equivalentierelatie $\sim_L$.

De tweede partitie heeft te maken met de DFA voor L. Voor elke toestand $q \in Q$ definiëren we een deelverzameling $reach(q)$ van Σ^* als volgt:

$$reach(q) = \{w \in \Sigma^* | \delta^*(q_s, w) = q\}$$

In woorden: $reach(q)$ is de verzameling strings die je van de begintoestand in toestand q brengen.

Een alternatieve manier om $reach(q)$ te verstaan: verander in de DFA elke eindtoestand naar een gewone toestand en duid dan enkel q aan als eindtoestand. Dan is $reach(q)$ de (reguliere) taal bepaald door die nieuwe DFA. (overtuig je

[8] In het engels coarse

daarvan!)

De verzameling $\{reach(q)|q \in Q\}$ is een partitie van Σ^* als elke toestand bereikbaar is (overtuig je daarvan!) en voldoet dan aan de volgende eigenschappen (verifieer en als je er niet uitgeraakt, zoek hulp):

- er zijn maar een eindig aantal verschillende $reach(q)$; we zeggen: de partitie is eindig
- $s \in \Sigma^*$ wordt aanvaard door de DFA alss $s \in reach(q)$ voor een $q \in F$
- als $s \in reach(q)$ dan $\forall a \in \Sigma : sa \in reach(\delta(q, a))$

De partitie $\{reach(q)|q \in Q\}$ bepaalt een equivalentierelatie $\sim_{DFA}$ op de gewone manier. In dit geval is dat het zelfde als schrijven:

Definitie 13.2 $x \sim_{DFA} y \iff \delta^*(q_s, x) = \delta^*(q_s, y)$

$\sim_{DFA}$ voldoet aan de volgende drie eigenschappen:

1. $\forall x, y \in \Sigma^*, a \in \Sigma : x \sim_{DFA} y \rightarrow xa \sim_{DFA} ya$
   we zeggen: $\sim_{DFA}$ is rechts congruent
2. $\sim_{DFA}$ verfijnt $\sim_L$, of met andere woorden: $x \sim_{DFA} y \rightarrow x \sim_L y$
   of nog: als $x \sim_{DFA} y$ dan zitten x en y beide in L of geen van beide
3. het aantal equivalentieklassen van $\sim_{DFA}$ is eindig; we zeggen: $\sim_{DFA}$ heeft een eindige index

In het vervolg zullen we de equivalentieklasse waartoe een element x behoort aanduiden met $x_\sim$, dus $x_\sim = \{y|x \sim y\}$

Elke equivalentie relatie die voldoet aan (1), (2) en (3) hiervoor noemen we een *Myhill-Nerode*[9] *relatie voor L*: dat heeft zin, want die drie eigenschappen verwijzen naar L. We schrijven: de equivalentierelatie is MN(L). Wat hiervoor is aangegeven is dat vertrekkend van een DFA die L accepteert, we een MN(L) relatie kunnen construeren op Σ^*.

[9] John Myhill en Anil Nerode

We zullen nu het omgekeerde doen: gegeven een taal L over Σ en een MN(L)-relatie op Σ^*, dan transformeren we de MN(L)-relatie in een DFA zodanig dat $L_{DFA} = L$. Dat toont aan dat L regulier is. We formuleren dat als stelling:

Stelling 13.3 Gegeven een taal L over Σ en een MN(L)-relatie $\sim$ op Σ^*, dan definieert $(Q, \Sigma, \delta, q_s, F)$ een DFA die L bepaalt, waarbij

- $Q = \{x_\sim | x \in \Sigma^*\}$ [elke equivalentieklasse is een toestand]
- $q_s = \epsilon_\sim$ [de starttoestand bereik je met ϵ]
- $F = \{x_\sim | x \in L\}$ [eindtoestand wordt bereikt door een string in L]
- $\delta(x_\sim, a) = (xa)_\sim$ [definitie van δ]

Bewijs Dat δ goed gedefinieerd is, kan je bewijzen door gebruik te maken van de rechtse congruentie van $\sim$.

Verder zijn alle ingrediënten van de DFA duidelijk, in het bijzonder ook dat Q (en F) slechts een eindig aantal toestanden bevat.

We moeten nog bewijzen dat $L_{DFA} = L$:
$x \in L_{DFA} \triangleq \delta^*(\epsilon_\sim, x) \in F \Longleftrightarrow x_\sim \in F \triangleq x \in L$

De overgang bekom je door met inductie op de lengte van de string x te bewijzen dat $\delta^*(\epsilon_\sim, x) = x_\sim$. ■

We hebben nu ook alle ingrediënten voor de volgende:

Stelling 13.4 De overgangen van DFA naar de MN relatie, en van daar naar een DFA, zijn elkaars inversen - op een DFA-isomorfisme na.

Bewijs Formuleer deze stelling beter en maak het bewijs. ■

Zelf doen: Bewijs dat twee isomorfe DFA's een identieke $\sim_{DFA}$ hebben.

13.3 Het supremum van twee MN(L) relaties.

Eerst moeten we inzien dat voor een gegeven reguliere taal L, er veel MN(L)-relaties bestaan: vermits we voor elke DFA die L bepaalt een MN(L)-relatie krijgen, en vermits twee DFA's isomorf zijn alss hun MN(L)-relaties identiek zijn[10], en vermits er oneindig veel (niet-isomorfe) DFA's bestaan die de taal L aanvaarden[11], zijn er oneindig veel verschillende MN(L)-relaties op Σ^*.
Gegeven een reguliere taal L en twee MN(L) relaties $\sim_1$ en $\sim_2$. We kunnen van die twee relaties het supremum beschouwen in de tralie van equivalentierelaties of partities. Het supremum $\sim_{sup}$ is de fijnste relatie die groffer is dan beide relaties en die dus bevat, namelijk:

Definitie 13.5 $x \sim_{sup} y$ is de transitieve sluiting van $(x \sim_1 y) \vee (x \sim_2 y)$

Noteer R voor de relatie $(x \sim_1 y) \vee (x \sim_2 y)$, en S voor de transitieve sluiting van R. De essentie van transitieve sluiting zit erin dat

$$S(x,y) \equiv \exists z_1, ..., z_n : R(x,z_1) \wedge R(z_1,z_2) \wedge ...R(z_n,y)$$

Stelling 13.6 Het supremum van twee MN(L)-relaties is ook een MN(L)-relatie: als $\sim_1$ en $\sim_2$ MN(L)-relaties zijn, dan is het supremum $\sim_{sup}$ van $\sim_1$ en $\sim_2$ ook een MN(L) relatie.

Bewijs

1. $x \sim_{sup} y \equiv \exists z_i : ((x \sim_1 z_1) \vee (x \sim_2 z_1)) \wedge ((z_1 \sim_1 z_2) \vee (z_1 \sim_2 z_2)) \wedge ...$
   $((z_n \sim_1 y) \vee (z_n \sim_2 y))$
   $\Longrightarrow \exists z_i : \forall a \in \Sigma : (xa \sim_1 z_1 a) \vee (xa \sim_2 z_1 a)...$
   $\Longrightarrow \forall a \in \Sigma : xa \sim_{sup} ya$ dus $\sim_{sup}$ is rechts congruent.

2. Stel $x \in L$ en $x \sim_{sup} y$ dan volgt daaruit:
   $\exists z_i : ((x \sim_1 z_1) \vee (x \sim_2 z_1)) \wedge ((z_1 \sim_1 z_2) \vee (z_1 \sim_2 z_2)) \wedge ...$
   $\Longrightarrow z_1 \in L \wedge \exists z_i : ((z_1 \sim_1 z_2) \vee (z_1 \sim_2 z_2)) \wedge ((z_2 \sim_1 z_3) \vee (z_2 \sim_2 z_3)) \wedge ...$
   $\Longrightarrow z_2 \in L \ \wedge \ \exists z_i : ((z_2 \sim_1 z_3) \vee (z_2 \sim_2 z_3)) \wedge ...$
   $\Longrightarrow y \in L$; idem voor $\overline{L}$ en dus $\sim_{sup}$ verfijnt $\sim_L$

[10] Kan je dat bewijzen?
[11] Idem

3. Het aantal equivalentieklassen voor $\sim_{sup}$ is niet hoger dan het aantal equivalentieklassen van de twee $\sim_i$ dus eindig.

■

Die constructie is interessant, want nu kunnen we gegeven twee DFA's die dezelfde taal bepalen, een *supremum* van die twee DFA's definiëren die terug een DFA geeft (die dezelfde taal bepaalt) en zeker niet meer toestanden heeft.

13.4 De minimale DFA

Neem twee minimale DFA's voor een taal L. We zullen bewijzen dat ze isomorf zijn. Beiden hebben hetzelfde aantal toestanden N - anders was één van de twee niet minimaal. Neem nu het supremum van die twee DFA's: je krijgt een DFA met opnieuw N toestanden, want meer kan niet, maar ook minder niet. In termen van de MN(L) relaties door de drie betrokken DFA's geïnduceerd betekent dat dat die drie relaties identiek zijn. In termen van de drie DFA's betekent het dat ze alle drie isomorf zijn.

Pas nu weten we zeker dat de minimisatieprocedure eindigt met een uniek resultaat (op isomorfisme na).

De karakterisatie van de MN(L) relatie die hoort bij de minimale DFA is redelijk eenvoudig:

$x \sim_{min} y \Longleftrightarrow \forall s \in \Sigma^*(xs \in L \Leftrightarrow ys \in L)$

In essentie zegt dit: als twee toestanden f-gelijk zijn, dan zijn ze indentiek.

Zelf doen: Bewijs dat dit correct is.

13.5 MN(L) gebruiken om te bewijzen dat een taal niet regulier is

Als voorbeeld nemen we $L = \{a^n b^n | n \in \mathbb{N}\}$.

Stel dat er een relatie $\sim$ bestaat die voldoet aan (1) en (2) van de MN(L) regels en afkomstig is van de minimale DFA voor L. We bewijzen dat (3) niet kan, t.t.z. dat het aantal equivalentieklassen niet eindig is.

Bekijk de equivalentieklassen van die relatie. Het is duidelijk dat voor elke n de string a^n in een andere klasse moet zitten, want voor elk ervan is er een ander aantal b's dat je in F brengt. Er zijn dus oneindig veel equivalentieklassen voor

$\sim$, en dus kan L niet regulier zijn.

In volgend hoofdstuk zien we een meer intuïtieve eigenschap van reguliere talen die eventueel toelaat om van een gegeven taal te beslissen dat ze niet regulier is: het pompend lemma dat bovendien ook analogen heeft voor andere klassen van talen. Nochtans kunnen de pompende lemma's niet gebruikt worden om te bewijzen dat een taal regulier is. Myhill-Nerode geven dus een positieve karakterisatie van RegLan en hebben daarmee een fantastische bijdrage geleverd!

13.6 Afsluiter over Myhill-Nerode

Ons doel in deze sectie was vooral om te komen tot het bewijs dat de minimale DFA uniek is (op een isomorfisme na). Voor de volledigheid vermelden we nog de originele stelling van Myhill-Nerode - en die kan je nu ook zelf bewijzen:

Stelling 13.7 Stelling van Myhill en Nerode
Laat $L \subseteq \Sigma^*$ een taal zijn over Σ. De volgende drie uitspraken zijn dan equivalent:

1. L is regulier
2. er bestaat een Myhill-Nerode relatie voor L
3. definieer $\sim$ op Σ^* als volgt:
   $$x \sim y \Longleftrightarrow \forall s \in \Sigma^* : (xs \in L \Leftrightarrow ys \in L);$$
   de relatie $\sim$ heeft een eindige index

Natuurlijk bestaan er uitbreidingen van minimizatie van DFA's en de Myhill-Nerode stelling naar NFA's, maar de situatie daar is niet zo eenvoudig. Voor NFA's bijvoorbeeld bestaat geen unieke minimale machine. Wil je meer weten? Kom eens langs.

13.7 De complexiteit van minimizatie en equivalentie

Een DFA minimalizeren kan in polynoomtijd, maar het minimalizeren van NFA's is NP-hard. Het testen van de equivalentie van reguliere expressies is zelfs PSPACE-hard. Dat kan je intuïtief argumenteren door te wijzen op het feit dat de conversie van NFA's naar DFA's het aantal toestanden exponentieel kan doen toenemen.

14 Het pompen van strings in reguliere talen

M.b.v. MN(L)-relaties hebben we ondertussen al bewezen dat RegLan niet alle talen bevat. We bekijken hier een tweede manier om van een taal aan te tonen ze niet regulier is.

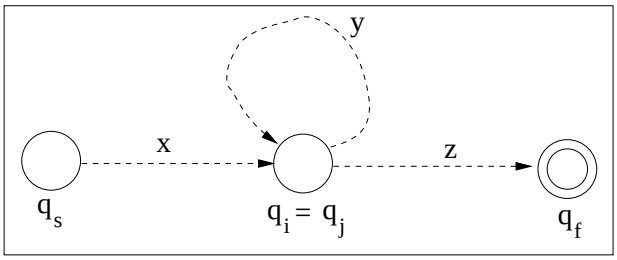
Neem een reguliere taal L met oneindig veel strings. Voor L bestaat een DFA. Die DFA heeft $N = \#Q$ toestanden. Neem een string in L die langer is dan N, en begin met die string in de hand op je tocht van de starttoestand naar een eindtoestand. Vermits er maar N toestanden zijn en je string meer dan N lang is en je bij elke overgang juist één symbool achterlaat, moet je op je weg naar de eindtoestand minstens één toestand S twee keer (of meer) tegenkomen: je hebt ergens een kring gemaakt. Tijdens die kring heb je een substring van de oorspronkelijke string gebruikt, t.t.z. je initiële string is van de vorm xyz, waarbij x, y en z substrings zijn, x het stuk voor je aan S kwam, y het stuk van S tot de eerstvolgende weer aan S, en z alles erna. Probeer je er nu van te overtuigen dat je die kring twee keer zal doen als je als initiële string $xyyz$ had gekregen, en dat $xyyz$ ook wordt aanvaard. En hetzelfde voor xz, en $xyyyz$ ... en xy^iz voor elke i.

Formeel nu:

Stelling 14.1 Het pompend lemma voor reguliere talen
Voor een reguliere taal L bestaat een pomplengte d, zodanig dat als $s \in L$ en $|s| \geq d$, dan bestaat er een verdeling van s in stukken x, y en z zodanig dat $s = xyz$ en

1. $\forall i \geq 0 : xy^iz \in L$
2. $|y| > 0$
3. $|xy| \leq d$

Bewijs Neem een DFA die L bepaalt. Neem $d = \#Q + 1$.
Neem een willekeurige $s = a_1a_2...a_n$ met $n \geq d$. Beschouw de accepterende sequentie van toestanden $(q_s = q_1, q_2, ..., q_f)$ voor s; die heeft lengte strikt groter dan d, dus zijn er bij de eerste d zeker twee toestanden gelijk (omdat er maar $d-1$ toestanden zijn). Stel dat q_i en q_j gelijk zijn met $i < j \leq d$ dan nemen we $x = a_1a_2...a_i$ en $y = a_{i+1}...a_j$ en z de rest van de string. Alles volgt nu direct. Figuur 2.16 illustreert de verdeling van s in xyz. ■



Figuur 2.16: Illustratie van de verdeling van de string

14.1 Het pompend lemma gebruiken

Het is belangrijk eerst voor jezelf uit te maken dat het pompend lemma niet nuttig is om te bewijzen dat een taal regulier is - probeer maar :-)

We kunnen het wel gebruiken om te bewijzen dat een gegeven taal niet regulier is. Als voorbeeld nemen we nog eens de taal $L = \{a^n b^n | n \in \mathbb{N}\}$ over het alfabet $\{a, b\}$.

Stel dat er voor die taal een pomplengte d bestond, beschouw dan de string $s = a^d b^d$. Neem een willekeurige opdeling van $s = xyz$ met $|y| > 0$. Dan zijn er drie mogelijkheden

- y bevat alleen a's: dan bevat xyyz meer a's dan b's en zit dus niet in L
- y is van de vorm $a^i b^j$ met $i \neq 0, j \neq 0$: dan bevat xyyz niet alle a's voor de b's, en zit dus niet in L
- y bevat alleen b's: dan bevat xz meer a's dan b's en zit dus niet in L

Bijgevolg kan L niet regulier zijn.

We hebben punt 3 van het lemma niet gebruikt. Als je dat wel wil doen, dan gaat het bewijs dat L niet regulier is soms korter of gemakkelijker. In dit geval wordt het:

Stel dat er voor die taal een pomplengte d bestond, beschouw dan de string $s = a^d b^d$. Neem een willekeurige opdeling van $s = xyz$ met $|y| > 0$ en $|xy| \leq d$. Dan bestaat y uitsluitend uit a's en dus bevat xz minder a's dan b's en zit dus niet in L.

Merk op: Het pompend lemma gebruiken om te bewijzen dat L niet in RegLan zit, heeft de volgende ingrediënten:

- voor een willekeurig getal d gekozen als pomplengte
- bestaat een string s langer dan d
- waarvoor **elke** opdeling pompen verhindert

Vooral dat laatste realiseren is belangrijk - hier een voorbeeld: neem L de taal gegenereerd door de reguliere expressie ab^*c. We gaan (verkeerdelijk) bewijzen dat L niet regulier is door (foutief) gebruik te maken van het pompend lemma. Neem een willekeurige pomplengte d (groter dan bijvoorbeeld 10), en neem een willekeurige string met lengte groter dan d+1. De string is van de vorm ab^ic met $i > 9$. Neem voor x,y en z uit de stelling $x = \epsilon$, $y = a$ en $z = b^ic$. Het is nu duidelijk dat xyyz niet behoort tot L en dus kunnen we de string niet pompen. Dus is L niet regulier ...

Zelf doen:

Welke fout werd hierboven gemaakt?

Definieer wat talen en gebruik het pompend lemma om na te gaan of ze regulier (kunnen) zijn. Is de taal van reguliere expressies regulier?

Bestaat er een taal waarvan elke string kan gepompt worden en die toch niet regulier is?

Bestaat er een minimale pomplengte? Is er een verband met de minimale DFA voor de taal?

15 Doorsnede, verschil en complement van DFA's

We hebben al wel de unie van reguliere talen bekeken, maar nog niet de andere gebruikelijke set-operaties: doorsnede, (symmetrisch) verschil en complement. Stel gegeven twee DFA's $(Q_i, \Sigma, \delta_i, q_{si}, F_i)$ voor i=1,2. We maken een generische product DFA $(Q, \Sigma, \delta, q_s, F)$ als volgt:

- $Q = Q_1 \times Q_2$
- $\delta(p \times q, x) = \delta_1(p, x) \times \delta_2(q, x)$
- $q_s = q_{s1} \times q_{s2}$

Nu moeten we enkel nog F bepalen om te komen tot een volledige definitie. Dat kan natuurlijk op verschillende manieren. Hier zijn er een aantal:

- $F = F_1 \times F_2$: de DFA is nu de doorsnede van de twee talen
- $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$: de DFA is nu de unie van de twee talen
- $F = F_1 \times (Q_2 \setminus F_2)$: de DFA bepaalt nu de strings die tot L_1 behoren, maar niet tot L_2
- $F = (Q_1 \setminus F_1) \times (Q_2 \setminus F_2)$: de DFA bepaalt nu de strings die tot geen van beide talen behoren

De bovenstaande constructies tonen aan dat de unie (dat wisten we al), de doorsnede, het verschil en het symmetrisch verschil van twee reguliere talen ook regulier is. Daaruit volgt ook dat het complement van een reguliere taal regulier is, want $\overline{L} = \Sigma^* \setminus L$.

Zelf doen:

Vind een eenvoudigere constructie van een complements-DFA als de DFA van een taal gegeven is.

Als diezelfde constructie wordt gedaan op een NFA, krijg je dan nog wat je bedoelde? Waarom (niet)?

16 Reguliere expressies en lexicale analyse

Met een reguliere expressie kan dikwijls gemakkelijk gespecifieerd worden welke *input* in een bepaalde context toegelaten is. Bijvoorbeeld:

$$20(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)$$

geeft aan dat ergens alleen een jaartal in deze eeuw mag ingetypt worden.

Voor programmeertalen wordt heel dikwijls van RE's gebruik gemaakt om het lexicon van de taal te definiëren. Een klein stukje van het lexicon van Java zou kunnen zijn: $(a|b|c)(a|b|c|0|1|2)^*$ | $(-|\epsilon)(1|2)(0|1|2)^*$ waarmee we dan identifiers kunnen beschrijven die met één van de letters a,b of c beginnen en dan nog een willekeurig aantal letters en cijfers (enkel 0,1,2) kunnen bevatten, en gehele getallen die optioneel een minteken hebben en dan ...

Het wordt snel omslachtig als we geen afkortingen gebruiken en het is dus gebruikelijk om zulke specificatie van het lexicon te doen als volgt:

$$PosCijfer \leftarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$
$$Cijfer \leftarrow PosCijfer \mid 0$$

en dan kan dat jaartal voorgesteld worden als

$$DezeEeuw \leftarrow 20CijferCijfer$$

en de algemene vorm van een integer als $(+|-|\epsilon)PosCijfer\ Cijfer^* \mid 0$

en voor een veld waarin een bankrekeningnummer moet ingetypt worden:

$$BANKNR \leftarrow Cijfer^3(-\mid\epsilon)Cijfer^7(-\mid\epsilon)Cijfer^2$$

Op die manier wordt het nu eenvoudig om een beschrijving te geven van het lexicon van bijvoorbeeld een programmeertaal. Hier is een stukje uit de beschrijving van Java:

$JavaProgr \leftarrow (Id|Int|Float|Op|Delimiter)^*$

$Id \leftarrow Letter\ (Letter|Cijfer)^*$

$Cijfer \leftarrow PosCijfer \mid 0$

$PosCijfer \leftarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$Teken \leftarrow (+|-|\epsilon)$

$Unsigned \leftarrow PosCijfer\ Cijfer^*$

$Int \leftarrow Teken\ Unsigned$

$Float \leftarrow Int\ .\ Unsigned$

...

Die beschrijving kan gebruikt worden om een lexer te maken voor Java, t.t.z. een programma dat aan lexicale analyse van een reeks tekens doet en beslist of ze behoren tot de taal (Java in dit geval). Er bestaan natuurlijk tools om gegeven een beschrijving van een lexicon m.b.v. reguliere expressies en afkortingen zoals hierboven, de benodigde automaten te genereren en de juiste glue-code om het zaakje werkende te houden. Zo vind je flex (zie `http://flex.sourceforge.net/`) dat C-code genereert: die C-code implementeert de DFA's en de glue-code. jflex (`http://www.jflex.de/`) is op hetzelfde principe gebaseerd en genereert Java-code. flex en jflex zijn lexicale analyse generatoren.

Zelf doen:

Lees meer over (j)flex.

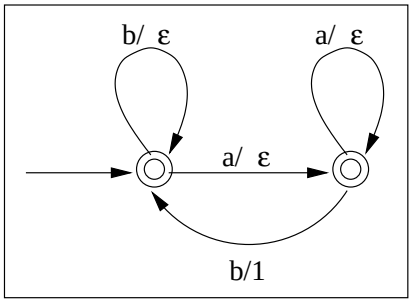
Probeer vanuit een reguliere expressie E een Prolog programma te genereren, met een predicaat lex/1 dat opgeroepen met een string s slaagt alss s tot L_E behoort: een string zoals abc stel je in Prolog voor door $[a, b, c]$. Een RE zoals $a^*b \mid c$ kan je voorstellen als de term $of([ster(a), b], [c])$ (maar kan ook anders).

De beschrijving van bijvoorbeeld *JavaProgr* hierboven, is een BNF-grammatica, anders gezegd *staat in Backus-Nauer vorm*: eigenlijk is die bedoeld voor context-vrije talen. BNF kan je ook gebruiken voor reguliere talen, omdat die ook context-vrij zijn: zie hoofdstuk 19.

17 Varianten van eindige toestandsautomaten

17.1 Transducer

Een transducer zet een string om in een andere: we passen de definitie van een DFA een beetje aan, zodat ook output kan geproduceerd worden. Eén figuur is bijna een definitie waard:



Figuur 2.17: Een eenvoudige transducer

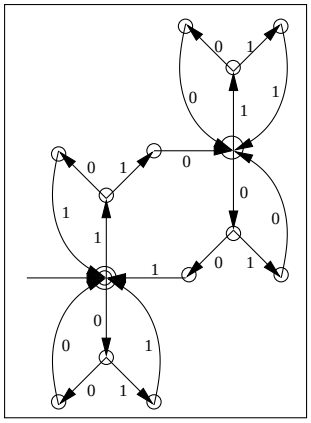
De labels zijn nu van de vorm a/x waarbij a in het inputalfabet zit en x in een outputalfabet (inbegrepen de lege string). Wat voor de / staat wordt gebruikt om de weg te vinden in de transducer alsof het een DFA was. Wat na de / staat wordt op de output gezet als die boog genomen wordt. Bovenstaande transducer accepteert elke string en geeft als output een 1 voor elke b die vlak na een a komt.

17.2 Een optelchecker m.b.v. een DFA

Een DFA kan alleen gebruikt worden om te beslissen of een string behoort tot een taal. Zo zou je kunnen een taal definiëren van strings die correcte optellingen voorstellen en als die taal regulier is er een DFA voor bouwen. Hier is een poging: $\Sigma = \{0, 1\}$. De twee getallen die we willen optellen en het resultaat komen in binair, omgekeerd en we maken ze even lang door bij de kortste(n) wat leidende nullen toe te voegen. Dus als we 3 willen optellen bij 13, met resultaat 16, dan hebben we de drie bitstrings 11000 10110 en 00001. Die mengen we nu systematisch, t.t.z. we maken groepjes van 3 bits die op de i-de plaats voorkomen en schrijven die groepjes achter elkaar. Dus we hebben

110 100 010 010 001

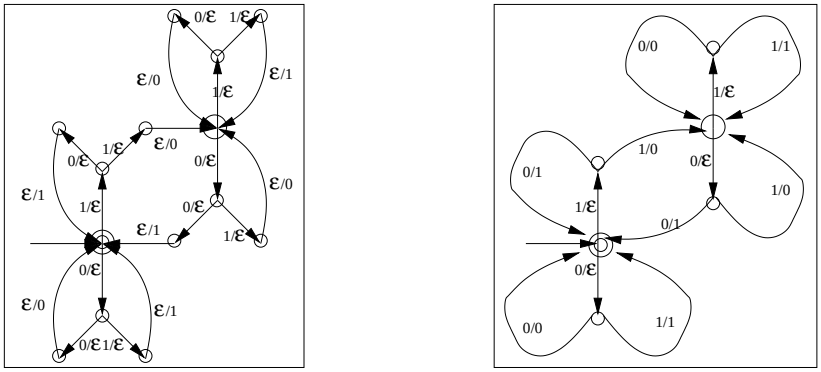
waarbij de blanco's enkel dienen om de groepering per drie te laten zien. Dus de string 110100010010001 stelt de correcte optelling 3+13=16 voor. Een DFA voor de taal van correcte optellingen staat in figuur 2.18.



Figuur 2.18: Een optelchecker

17.3 Een opteller m.b.v. een transducer

Optellen bestaat eigenlijk in: gegeven twee getallen als input, output de som. Dat kan met een transducer: gebruik dezelfde voorstelling van de twee getallen die je wil optellen als hiervoor, en meng die op dezelfde manier, dus 3+13 wordt voorgesteld als de string 1110010100. Figuur 2.19 toont twee optel-transducers die van de optelchecker zijn afgeleid.



Figuur 2.19: Twee transducers die als output de som geven

Zelf doen:

Kan je met een transducer ook andere rekenkundige bewerkingen doen, bijvoorbeeld min of maal?

Is de output van een transducer ook een reguliere taal?

Kan je een transducer schrijven die elk derde teken uit een inputstring geeft?

Kan je een transducer schrijven die elk derde teken uit een inputstring niet geeft?

Kan je een transducer schrijven die inputstrings omgekeerd uitschrijft?

Kan je elke reguliere taal als output van een transducer krijgen?

17.4 Two-way finite automata

Een DFA wordt ook wel een *one-way finite automaton* genoemd. De reden is dat je hem ook kan beschrijven als een machine met een invoerband waarop de inputstring staat. Die machine heeft een leeskop die in de begintoestand op het eerste teken staat, en bij elke overgang één positie naar rechts opschuift in de input: altijd in dezelfde richting, namelijk naar het einde van de string toe.

Het is slechts een kleine aanpassing aan de automaat om toe te laten dat de leeskop in twee richtingen mag bewegen: daarvoor pas je de definitie van δ gepast aan (zie in hoofdstuk 3 hoe dat voor de Turingmachines gebeurt). Je krijgt dan een 2DFA.

Van andere klassen van automaten is het geweten dat meer vrijheid laten in de manipulatie van het *geheugen* aanleiding geeft tot meer berekeningskracht: om dit te appreciëren moeten we daarmee eerst nog kennismaken natuurlijk, maar denk eraan bij de PDA en de LBA.

De vraag *is een 2DFA krachtiger dan een DFA*, of *zijn er niet-reguliere talen die met een 2DFA kunnen herkend worden* is dus van belang. Het blijkt dat de 2DFA ook enkel de reguliere talen bepaalt.

17.5 Büchi automaten

Büchi[12] automaten trekken op NFA's maar je geeft er geen eindige strings aan, wel oneindig lange: er is dus geen moment waarop je string *op* is bij het doorlopen van de automaat en de definitie van welke strings geaccepteerd worden door de Büchi automaat moet worden aangepast. Die definitie wordt:

Definitie 17.1
Een oneindige string s wordt aanvaard door een Büchi automaat indien de rij toestanden waarlangs je passeert oneindig dikwijls een aanvaardende toestand heeft.

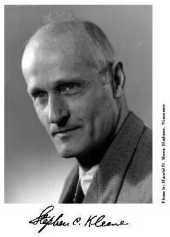
Je kan dat natuurlijk niet uitproberen, maar daarvoor dienen Büchi automaten niet noodzakelijk: je modelleert er een probleem mee (bijvoorbeeld een oneindig repetitief process, een protocol ...) en bewijst dan voor welke strings de acceptance conditie waar is.

Niet-deterministische Büchi automaten zijn strikt sterker dan deterministische Büchi automaten: als voorbeeld kan je voor taal $(a|b)^*b^\omega$ eens een Büchi automaat proberen te maken.

[12] Julius Büchi

18 Referenties

Veel van wat je hierboven leerde is afkomstig van Stephen Kleene, die je ook al kende van een fixpoint stelling. Hij heeft ook bijdragen geleverd in logica, recursie theorie en intuïtionisme.



In moderne teksten over reguliere talen/expressies/machines is de volgorde niet altijd dezelfde, maar de vorige hoofdstukken hebben aangetoond dat het kip&ei probleem niet bestaat: ze zijn equivalent. Ook zijn er dikwijls kleine verschillen in de basisdefinities, maar ook die maken niks uit: bijvoorbeeld of δ totaal is of niet, of er meer dan één eindtoestand is ...

Eenzelfde diversiteit in aanpak/definities vinden we later ook terug bij andere talen/machines en we proberen oog te hebben voor die verschillen, maar de equivalenties ervan in te zien.

Referenties:

- Dexter C. Kozen *Automata and Computability*
- Peter Linz *An Introduction to Formal Languages and Automata*
- Michael Sipser *Introduction to the Theory of Computation*
- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman *Introduction to Automata Theory, Languages, and Computation*
- Marvin Lee Minsky *Computation: Finite and Infinite Machines (Automatic Computation)*

Als je hierdoor begeesterd wordt, vergeet dan ook niet een andere standaard referentie: *Regular algebra and finite machines*[13] van John Horton Conway - dezelfde die *The Game of Life* uitvond dat later in deze tekst nog aan bod komt, en nog zoveel andere boeiende zaken (o.a. het *Angels and Devils game*) die aan ons inzicht in algoritmiek rechtsreeks aanbelangen.

[13] Dit werk staat niet in onze bib, maar als je geïnteresseerd bent, kom eens langs.

Zelf doen:
Als L1 en L2 reguliere talen zijn, is L3 dan ook regulier? We spreken af dat het alfabet altijd hetzelfde is en minstens a en b bevat. We gebruiken de notatie $\hat{s}$ om de string aan te duiden die dezelfde tekens als s bevat maar in omgekeerde volgorde. Schrijf telkens formeel neer wat L3 is.

L3 is strings van gelijke lengte uit L1 en L2 gemengd (schrijf formeel neer wat gemengd is)

L3 = $\widehat{L1}$

L3 is strings van even lengte uit L1

L3 is strings s van even lengte uit L1 en die dan in twee gekapt s1s2 met gelijke lengte en dan s1 concat omgekeerde van s2

L3 is strings s van L1 en dan s concat omgekeerde van s

L3 is strings van L1 met elke a vervangen door b

L3 is alles uit L1 dat niet in L2 zit

L3 is de taal van strings uit L1 waarin de symbolen op de even plaatsen weggenomen zijn

$L3 = \{x | \exists y \in L2, xy \in L1\}$

$L3 = \{x | \exists y \in L2, yx \in L1\}$

19 Contextvrije talen en hun grammatica

Reguliere expressies geven een manier om een taal te bepalen. Een kleine uitbreiding aan RE's is het toelaten van *afkortingen* voor RE's om op die manier kortere beschrijvingen van een taal te verkrijgen. Een afkorting bestaat erin van een naam te geven aan een ding en als je op een bepaalde plaats de naam gebruikt hebt, mag je daar het ding zelf zetten (of een kopie ervan) en dan is de betekenis nog dezelfde en de naam is weg. Maar bijvoorbeeld in

$$\text{Haakjes} \rightarrow \text{HaakjesHaakjes} \mid [\text{Haakjes}] \mid \epsilon$$

kan je het symbool Haakjes niet kwijtgeraken door substitutie. Toch kunnen we die regel gebruiken om een taal te genereren. Zulke regels vormen een *contextvrije grammatica*. Voor we een definitie geven, eerst nog wat voorbeelden:

Voorbeeld

- $S \rightarrow aSb$

  $S \rightarrow \epsilon$

  Deze grammatica beschrijft strings van de vorm $a^n b^n$

- $S \rightarrow PQ$

  $P \rightarrow aPb$

  $P \rightarrow \epsilon$

  $Q \rightarrow cQd$

  $Q \rightarrow \epsilon$

  Deze grammatica beschrijft strings van de vorm $a^n b^n c^m d^m$

- $Stat \rightarrow Assign$

  $Stat \rightarrow ITE$

  $ITE \rightarrow If\ Cond\ Then\ Stat\ Else\ Stat$

  $ITE \rightarrow If\ Cond\ Then\ Stat$

  $Cond \rightarrow Id == Id$

  $Assign \rightarrow Id := Id$

  $Id \rightarrow a$

  $Id \rightarrow b$

  $Id \rightarrow c$

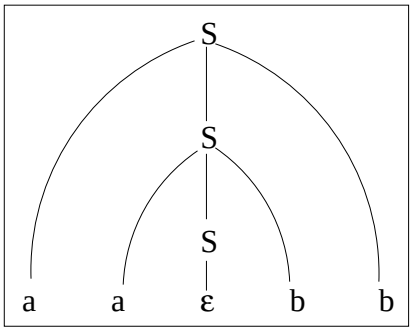
Informeel kunnen we zeggen: een contextvrije grammatica (CFG) bestaat uit regels; aan de linkerkant van een regel staat een niet-eindsymbool; aan de rechterkant staat een opeenvolging van eindsymbolen, niet-eindsymbolen en ϵ . Er is een startsymbool.

Je kan een CFG gebruiken om strings te genereren, en om na te gaan of een string voldoet aan de grammatica. Genereren doe je door een afleiding te maken vertrekkend van het startsymbool. Voor het eerste voorbeeld is wat volgt een afleiding:

$$S \rightarrow aSb \rightarrow aaSbb \rightarrow aabb$$

De idee achter afleiding is: in een string waarin nog een niet-eindsymbool staat, kies een niet-eindsymbool X en vervang X door de rechterkant van een regel waarin X links voorkomt. Begin met het startsymbool en werk door tot er alleen nog eindsymbolen staan.

We kunnen die afleiding ook voorstellen in een *syntax boom* of *parse tree*: zie figuur 2.20.



Figuur 2.20: Syntaxboom van aabb

Definitie 19.1 Contextvrije grammatica - CFG
Een contextvrije grammatica is een 4-tal (V, Σ, R, S) waarbij

- V een eindige verzameling niet-eindsymbolen is (ook variabelen genoemd, of non-terminals)
- Σ een eindig alfabet van eindsymbolen (terminals), disjunct met V
- R is een eindige verzameling regels (of producties); een regel is een koppel van één niet-eindsymbool en een string van elementen uit $V \cup \Sigma_\epsilon$; we schrijven de twee delen van zulk een koppel met een $\rightarrow$ ertussen
- S is het startsymbool en behoort tot V

Dikwijls ligt het voor de hand welke de eindsymbolen zijn en welk symbool het startsymbool is: we geven dan alleen de regels.

Definitie 19.2 Afleiding m.b.v. een CFG
Gegeven een CFG (V, Σ, R, S). Een string f over $V \cup \Sigma_\epsilon$ wordt afgeleid uit een string b over $V \cup \Sigma_\epsilon$ m.b.v. de CFG als er een eindige rij strings $s_0, s_1, ..., s_n$ bestaat zodanig dat

- $s_0 = b$
- $s_n = f$
- s_{i+1} verkregen wordt uit s_i (voor $i < n$) door in s_i een niet-eindsymbool X te vervangen door de rechterkant van een regel waarin X links voorkomt.

We noteren: $s_i \Rightarrow s_{i+1}$ en $b \Rightarrow^* f$

Definitie 19.3 Taal bepaald door een CFG
De taal L_{CFG} bepaald door een CFG (V, Σ, R, S) is de verzameling strings over Σ die kunnen afgeleid worden van het startsymbool S; of meer formeel: $L_{CFG} = \{s \in \Sigma^* | S \Rightarrow^* s\}$.

Definitie 19.4 Contextvrije taal - CFL
Een taal L is **contextvrij** indien er een CFG bestaat zodanig dat $L = L_{CFG}$

We zullen i.v.m. contextvrije talen zowat dezelfde weg bewandelen als voor reguliere talen: we definiëren een machine die contextvrije talen bepaalt en we bewijzen dat die machines *equivalent* zijn met de contextvrije grammatica's; we bestuderen het verschil tussen deterministische en niet-deterministische versies van die machines; we maken een versie van het pompend lemma voor contextvrije talen; we bestuderen de algebraïsche operaties op de contextvrije talen. We zullen niet aan minimizatie doen. We zullen aandacht besteden aan ambiguïteit: dit probleem hebben we niet bij reguliere talen besproken omdat het daar niet belangrijk is.

19.1 Ambiguïteit

We nemen als voorbeeld de CFG Arit1

- Expr → Expr + Expr
- Expr → Expr * Expr
- Expr → a

waarbij Expr het startsymbool is. Neem de string $a + a * a$: die zit duidelijk in de taal bepaald door de Arit1. Bekijk nu de twee afleidingen van die string (we onderlijnen de gesubstitueerde non-terminal als er keuze is):

$$Expr \Rightarrow \underline{Expr} + Expr \Rightarrow a + Expr \Rightarrow a + \underline{Expr} * Expr$$
$$\Rightarrow a + a * Expr \Rightarrow a + a * a$$

en

$$Expr \Rightarrow Expr + \underline{Expr} \Rightarrow Expr + Expr * \underline{Expr} \Rightarrow Expr + \underline{Expr} * a$$
$$\Rightarrow Expr + a * a \Rightarrow a + a * a$$

In de eerste afleiding hebben we telkens het meest linkse niet-eindsymbool verder ontwikkeld, en in de tweede afleiding het meest rechtse. Maar als we die afleidingen omzetten naar een parse tree krijgen we twee keer hetzelfde. Die afleidingen zijn dus niet essentieel verschillend en dikwijls zullen we daarom ook enkel kijken naar meest-linkse[14] afleidingen.

[14] In het engels: leftmost derivation.

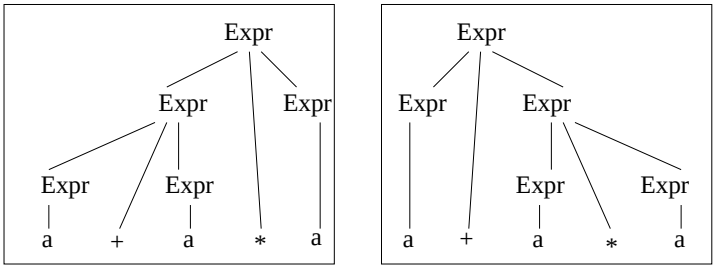
Beschouw nu de volgende twee meest-linkse afleidingen van $a + a * a$:

$$Expr \Rightarrow Expr + Expr \Rightarrow a + Expr \Rightarrow a + Expr * Expr \Rightarrow^* a + a * a$$

en

$$Expr \Rightarrow Expr * Expr \Rightarrow Expr + Expr * Expr \Rightarrow a + Expr * Expr \Rightarrow^* a + a * a$$

of in termen van de overeenkomstige parse trees: zie figuur 2.21:



Figuur 2.21: Twee parse trees voor $a + a * a$

De string a+a*a heeft meerdere meest-linkse afleidingen en dus ook meerdere parse trees: de string wordt daarom *ambigu* genoemd. We noemen de grammatica Arit1 ambigu omdat er strings bestaan in L_{Arit1} die ambigu zijn t.o.v. Arit1. A priori is het niet duidelijk of voor dezelfde taal ook een niet-ambigue grammatica bestaat. Eerst een nuttige definitie:

Definitie 19.5 Equivalente CFG's
Twee contextvrije grammatica's $CFG1$ en $CFG2$ zijn equivalent indien
$L_{CFG1} = L_{CFG2}$

Je moet eigenlijk bewijzen dat dit een equivalentierelatie op de CFG's definieert, maar dat is natuurlijk kinderspel voor jullie.

Hier is een andere CFG Arit2:

- Expr $\rightarrow$ Expr + Term
- Expr $\rightarrow$ Term
- Term $\rightarrow$ Term $*$ a
- Term $\rightarrow$ a

Je kan nagaan dat Arit1 en Arit2 equivalent zijn.

Je kan ook nagaan dat $a + a * a$ nu enkel de meest-linkse afleiding

$$Expr \Rightarrow Expr + Term \Rightarrow Term + Term \Rightarrow a + Term$$
$$\Rightarrow a + Term * a \Rightarrow a + a * a$$

heeft en dat elke string slechts één parse tree heeft voor Arit2. Daarom noemen we Arit2 een niet-ambigue grammatica.

Niet elke contextvrije taal heeft een niet-ambigue CFG: zulk een contextvrije taal heet *inherent ambigu*. Hier is een voorbeeld van een inherent ambigue taal: $\{a^n b^n c^m, a^n b^m c^m | n, m \geq 0\}$. Het bewijs daarvan kan je vinden in het boek van Hopcroft-Motwani-Ullman, maar bewijs zelf dat de taal contextvrij is! Later hebben we het nog over ambiguïteit.

Als een niet-terminaal symbool aan de linkerkant van meerdere regels voorkomt, dan kunnen we die samennemen. Bijvoorbeeld kan grammatica Arit2 ook beschreven worden als:

- Expr $\rightarrow$ Expr + Term | Term
- Term $\rightarrow$ Term $*$ a | a

19.2 CFG's in een speciale vorm

Soms is het handig een meer restrictieve vorm van een grammatica te hebben, liefst zonder contextvrije talen uit te sluiten. Hier is zulk een vorm:

Definitie 19.6 Chomsky Normaal Vorm
Een CFG heeft de Chomsky Normaal Vorm als elke regel één van de volgende vormen heeft:

1. A $\rightarrow$ BC
2. A $\rightarrow \alpha$
3. S $\rightarrow \epsilon$

Daarin is α een eindsymbool, A is een niet-eindsymbool, en B en C zijn niet-eindsymbolen verschillend van S (het startsymbool).

Soms laat men de regel S $\rightarrow \epsilon$ niet toe: dan kan die CFG niet de lege string afleiden natuurlijk.

Stelling 19.7 Voor elke contextvrije grammatica bestaat een equivalente contextvrije grammatica in Chomsky Normaal Vorm.

Bewijs We geven een constructief bewijs: het is lang maar niet moeilijk. We vertrekken van een willekeurige CFG en transformeren hem terwijl we equivalentie bewaren naar Chomsky Normaal Vorm.

1. We beginnen met te zorgen dat er een startsymbool is dat alleen links in een regel voorkomt: als S het startsymbool is in de grammatica, vervang het dan overal door een nieuw niet-eindsymbool (bijvoorbeeld X) en voeg de regel $S \rightarrow X$ toe

2. Daarna voldoen we aan de derde eis van Chomsky Normaal Vorm:

stel dat we een regel $\mathcal{E} = A \rightarrow \epsilon$ hebben en een regel $\mathcal{R} = B \rightarrow \gamma$ waarin A voorkomt in γ, dan definiëren we de verzameling regels $V(\mathcal{E}, \mathcal{R})$ als de verzameling regels van de vorm $B \rightarrow \eta$ waarbij η verkregen wordt uit γ door elke combinatie van voorkomens van A in γ weg te laten.

We transformeren de grammatica dan als volgt:

> zolang er regels $\mathcal{E} = A \to \epsilon$ en een regel $\mathcal{R} = B \to \gamma$ waarin A voorkomt zijn zodanig dat $V(\mathcal{E}, \mathcal{R})$ nieuwe regels bevat, voeg $V(\mathcal{E}, \mathcal{R})$ toe aan de grammatica

Zorg dat je inziet dat dit eindigt!

Daarna (dus ook nadat je het ingezien hebt :-) verwijderen we uit de bekomen grammatica alle regels van de vorm $A \to \epsilon$, behalve als $A = S$: dit mag de enige regel zijn die ϵ afleidt.

Zorg dat je inziet dat de bekomen grammatica nog dezelfde taal bepaalt: je kan dat doen door te redeneren op afleidingen.

3. Nu willen we afgeraken van de regels van de vorm $A \to B$. Voor een regel van de vorm $\mathcal{E} = A \to B$ en een regel van de vorm $\mathcal{R} = B \to \gamma$, definieer de regel $U(\mathcal{E}, \mathcal{R}) = A \to \gamma$.

> zolang er regels van de vorm $\mathcal{E} = A \to B$ (waarin B ook een niet-terminal is) en $\mathcal{R} = B \to \gamma$ bestaan en $U(\mathcal{E}, \mathcal{R})$ een nieuwe regel is, voeg $U(\mathcal{E}, \mathcal{R})$ toe aan de grammatica

Zorg dat je inziet dat dit eindigt!

Daarna (dus ook nadat je het ingezien hebt :-) verwijderen we uit de bekomen grammatica alle regels van de vorm $A \to B$.

Zorg dat je inziet dat de bekomen grammatica nog dezelfde taal bepaalt: je kan dat doen door te redeneren op afleidingen.

4. We hebben nu nog 3 soorten regels te behandelen:

(a) $A \to \gamma$ waar γ uit juist twee niet-terminalen bestaat: die laten we gerust

(b) $A \to \gamma$ waar γ minstens twee symbolen bevat: vervang elke terminal a door een nieuwe niet-terminaal A_a en voeg de regel $A_a \to a$ toe

(c) eventueel $S \to \epsilon$: die mag blijven

Zorg dat je inziet dat dit eindigt en dezelfde taal genereert!

5. de regels van de vorm $A \to X_1X_2...X_n$ met $n > 2$ vervang je door $A \to X_1Y_1$, $Y_1 \to X_2Y_2$, ..., $Y_{n-2} \to X_{n-1}X_n$

Zorg dat je inziet dat dit eindigt en dezelfde taal genereert!

Is aan alle eisen voldaan nu? ■

De Chomsky normaal vorm heeft als voordeel dat we direct kunnen zien of de taal van een grammatica de lege string bevat, en dat elke parse tree (bijna) een volledige binaire boom is: dat zal van pas komen in het pompend lemma voor CFL's. Bovendien heeft elke afleiding van een string van lengte $n > 0$ lengte $2n - 1$.

Er bestaan nog een andere normaalvormen voor CFG's: bijvoorbeeld de Greibach[15] normaal vorm laat enkel regels toe van de vorm

- A → aX
- S → ϵ

waarin X een (mogelijk lege) sequentie is van niet-terminalen, en a een terminal. Het voordeel van die vorm is dat een afleiding van een string van lengte n niet langer kan zijn dan n. Als oefening kan je ook het analoog van de stelling op pagina 61 bewijzen voor de Greibach normaal vorm.

Zelf doen:

Bewijs dat een afleiding van een string van lengte $n > 0$ uit een grammatica in Chomsky normaalvorm lengte $2n - 1$ heeft.

Bewijs dat een afleiding van een string van lengte $n > 0$ uit een grammatica in Greibach normaalvorm lengte n heeft.

Kan je de transformatie naar Chomsky normaalvorm gebruiken om in te zien dat elke (zuiver) Prologprogramma kan getransformeerd worden naar een equivalent Prologprogramma met in elke clause juist nul of twee doelen?

[15] Sheila Greibach

20 De push-down automaat

Een FSA heeft behalve zijn toestand geen geheugen. Doordat er maar een eindig aantal toestanden zijn, kan een FSA dus niet onbegrensd *tellen* en niet al te veel talen bepalen. Een push-down automaat heeft een onbeperkt geheugen, doch het kan slechts op een beperkte manier gebruikt worden: het geheugen is georganiseerd als een stapel (of stack) en daarvan kan op elk ogenblik enkel de top geïnspecteerd worden. Verder kunnen er elementen worden bijgezet of afgehaald. Net zoals bij de reguliere automaten gebruiken we een PDA om van een string na te gaan of ie geaccepteerd wordt of niet: dat gebeurt door opeenvolgende tekens van de string te consumeren en afhankelijk van de huidige toestand en dat teken, naar een andere toestand over te gaan. Bij de PDA komt daar nog bij dat die overgang kan afhangen van de top van de stack, en dat er een teken kan bijgezet worden op de stack. Net zoals bij de reguliere automaten definiëren we de PDA direct met niet-determinisme.

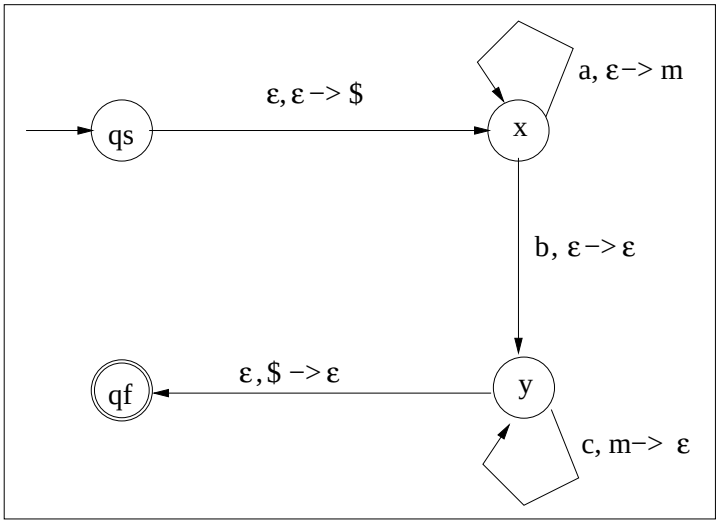
Definitie 20.1 Een push-down automaat
Een push-down automaat is een 6-tal $(Q, \Sigma, \Gamma, \delta, q_s, F)$ waarbij

- Q is een eindige verzameling toestanden
- Σ is een eindig inputalfabet
- Γ is een eindig stapelalfabet
- δ is een overgangsfunctie met signatuur $Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$
- q_s is de starttoestand
- $F \subseteq Q$ is een verzameling eindtoestanden

Deze definitie zegt nog niks over de werking. Eerst een grafische voorstelling van een PDA, in de stijl van de grafische voorstelling van een FSA. Als voorbeeld nemen we de PDA $(Q, \Sigma, \Gamma, \delta, q_s, F)$ met

- $Q = \{q_s, q_f, x, y\}$
- $F = \{q_f\}$
- $\Sigma = \{a, b, c\}$
- $\Gamma = \{m, \$\}$

De δ van de PDA kan je zien in de grafische voorstelling ervan in figuur 2.22.



Figuur 2.22: Een push-down automaat

De bedoeling van een label zoals $\alpha, \beta \to \gamma$ op een boog is

- indien α het eerste symbool is van de huidige string
- en β staat op de top van de stapel
- volg dan de boog en
  - verwijder α van de string
  - verwijder β van de stapel
  - zet γ op de stapel

Daarin mogen α, β en γ ook ϵ zijn, wat voor α wil zeggen: houd geen rekening met de huidige inputstring; voor β: houd geen rekening met de huidige top van de stapel; voor γ: push niets.

De bogen met hun labels specificeren dus de δ.
Als we nu beginnen in q_s met de string a^2bc^2, dan bevat de stapel $mm\$$ op het ogenblik dat we de eerste keer in y komen, en de string bestaat nog uit c^2. Na nog twee keer δ toepassen is de string helemaal geconsumeerd en nog één toepassing brengt ons in de eindtoestand. We zeggen: a^2bc^2 wordt geaccepteerd door de PDA, of a^2bc^2 behoort tot de taal door de PDA bepaald. Hier is een meer formele definitie van

Definitie 20.2 Aanvaarding van een string s door een PDA
Een string s wordt aanvaard door een PDA indien s kan worden opgesplitst in delen w_i, i= 1..m ($w_i \in \Sigma_\epsilon$), er toestanden q_j, j= 0..m zijn, en stacks $stack_k$, k=0..m ($stack_k \in \Gamma^*$), zodanig dat

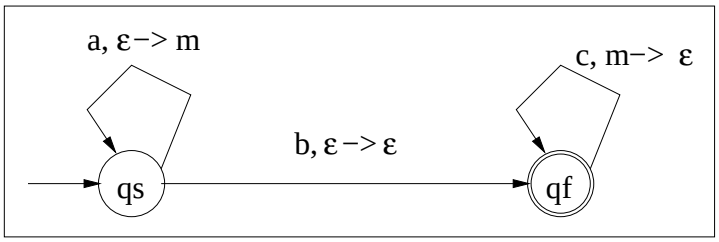
- $stack_0 = \epsilon$ (de stack is leeg in het begin)
- $q_0 = q_s$ (we vertrekken in de begintoestand)
- $q_m \in F$ (we komen aan in een eindtoestand met een lege string)
- $(q_{i+1}, y) \in \delta(q_i, w_{i+1}, x)$ waarbij $x, y \in \Gamma_\epsilon$ en
  $stack_i = xt, \quad stack_{i+1} = yt$ met $t \in \Gamma^*$

De laatste bullet geeft aan dat de overgangen juist gebeuren volgens δ.

Let op: Bovenstaande definitie zegt niets over de stackinhoud op het ogenblik dat we in de eindtoestand komen: die zegt enkel dat we met een lege string in een eindtoestand moeten geraken. Er bestaan alternatieve definities van PDA: soms mag meer dan één symbool gepusht worden in één overgang. Soms wordt acceptatie gedefinieerd als: *de string is op en de stack is leeg* (dan is F zelfs van geen belang) en soms als *de string is op en de stack is leeg en we zitten in een eindtoestand*. Uiteindelijk zijn al deze definities equivalent wat betreft de talen die kunnen bepaald worden. Bijkomende eisen kunnen zijn: in elke overgang wordt ofwel een symbool gepusht ofwel gepopt maar niet beide, of er is slechts één eindtoestand ... maar ook dat verandert niets essentieels.

Definitie 20.3 Taal bepaald door een PDA
De taal L bepaald door een PDA bestaat uit alle strings die door de PDA aanvaard worden.

Het is duidelijk dat de taal $\{a^n bc^n | n \geq 0\}$ door een PDA bepaald wordt: figuur 2.22 gaf een implementatie. Strings zoals aabc worden niet aanvaard.
Hoe tricky het is, zie je in figuur 2.23: op het eerste zicht zou je kunnen denken dat die dezelfde taal aanvaardt, maar is dat wel zo?



Figuur 2.23: Een tweede push-down automaat

Als je voor de PDA in figuur 2.23 acceptatie definieert als *aankomen in de eindtoestand met lege stack*, dan bepaalt deze PDA wel dezelfde taal: de details van de definitie bepalen dus wel wat een bepaalde PDA juist accepteert, maar niet de klasse van talen die door een PDA kunnen bepaald worden.

21 Equivalentie van CFG en PDA

We willen de volgende stelling bewijzen:

Stelling 21.1 Elke push-down automaat bepaalt een contextvrije taal en elke contextvrije taal wordt bepaald door een push-down automaat.

Bewijs Er zijn duidelijk twee delen in deze stelling. Het eerste deel bewijzen we in het lemma op pagina 70, het tweede in het lemma op pagina 72. ■

Eerst wat voorbereidend werk en een voorbeeld.

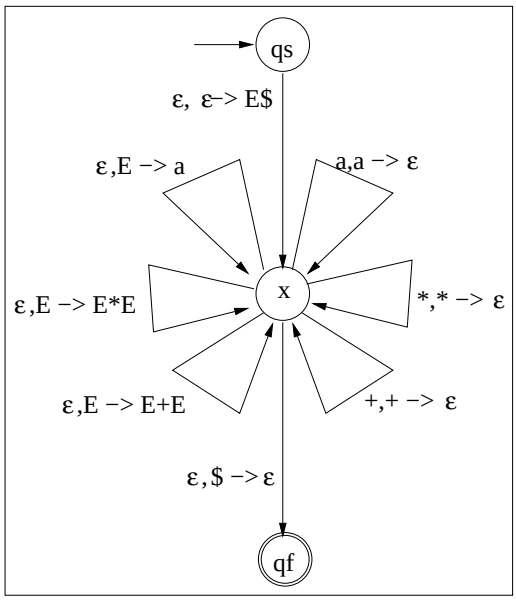
We hebben voordien al eens gezegd dat in één push er meerdere symbolen bij mogen komen op de stapel zonder dat dat de kracht van PDA's verandert (als je het nog niet deed ... dit is een goed moment om het te bewijzen). We gaan daarvan gebruik maken, omdat het de beschrijving heel wat korter maakt.

Voorbeeld *Beschouw de CFG Arit1 van pagina 58*

- $E \to E + E \qquad E \to E * E \qquad E \to a$

waarbij E het startsymbool is. En kijk nu eens naar de PDA in figuur 2.24. Die PDA vertoont wat meer symmetrie dan je vertrekkend van een andere grammatica zou krijgen, maar hij is op de volgende manier systematisch geconstrueerd:

- *er zijn slechts 3 toestanden: de begintoestand q_s, de eindtoestand q_f en één hulptoestand x*
- *er is slechts één boog van q_s naar x: die kijkt niet naar de string of de stapel, en zet een marker \$ op de stapel en het beginsymbool*
- *er is slechts één boog van x naar q_f: die consumeert niks van de string en haalt de marker \$ van de stapel*
- *de andere bogen gaan van x naar x; de labels corresponderen met*
  - *de symbolen uit het invoeralfabet: voor elke $\alpha \in \Sigma$, is er een boog met label $\alpha, \alpha \to \epsilon$; die bogen betekenen dus: als de top van de stapel gelijk is aan het eerste symbool van de string, consumeer dan beide*
  - *de regels van de grammatica: voor elke regel $X \to \gamma$ is er een label $\epsilon, X \to \gamma$; die bogen betekenen dus: als de top van de stapel een niet-eindsymbool X is, vervang het door de rechterkant γ van een regel in*



Figuur 2.24: Een PDA afgeleid van Arit1

de grammatica waarvan X de linkerkant is; γ is een rij eind- en niet-eindsymbolen

Zelf doen: Wat is de relatie tussen het inputalfabet en het stapelalfabet enerzijds en de terminals en non-terminals van de grammatica anderzijds?

We gebruiken die PDA eens om de string $a + a * a$ te parsen: tabel 2.3 laat de stapel en de string zien in de opeenvolgende toestanden. Merk op dat als we een reeks symbolen XYZ op de stapel willen pushen, we die eigenlijk in omgekeerde volgorde op de stapel willen hebben, en de representatie van een stapel is dan ook een string die we van voor aanvullen (zonder omkeren van XYZ) en waarvan we van voor weer wegnemen.

string	stapel	toestand
a+a∗a	ϵ	q_s
a+a∗a	E$	x
a+a∗a	E∗E$	x
a+a∗a	E+E∗E$	x
a+a∗a	a+E∗E$	x
+a∗a	+E∗E$	x
a∗a	E∗E$	x
a∗a	a∗E$	x
∗a	∗E$	x
a	a$	x
ϵ	$	x
ϵ	ϵ	q_f

Tabel 2.3: Parsing van $a + a * a$

Deze parsing geeft een meest-linkse afleiding. Die is niet noodzakelijk uniek: als je bij de tweede overgang E+E kiest i.p.v. E∗E dan kom je er ook nog!

Kan het zijn dat je vastloopt? Probeer eens om in de derde overgang E*E te kiezen i.p.v. E+E en je ziet het. Is dat erg?

Ga nu na dat elke string die door de Arit1 wordt bepaald, door de PDA in figuur 2.24 wordt aanvaard en omgekeerd.

Dat voorbeeld generaliseren we direct tot het volgende lemma:

Lemma 21.2 De constructie van een PDA uit een CFG zoals hierboven gegeven, levert een PDA die de taal L_{CFG} accepteert.

Bewijs Je kan nagaan dat er een één-éénduidig verband is tussen een afleiding in de CFG van een string s en een accepterende uitvoering van de PDA voor s. ■

Tenslotte nog een woordje over niet-determinisme en ambiguïteit: met bovenstaande constructie van een PDA uit een CFG bekomen we een niet-deterministische PDA indien er voor minstens één niet-terminaal symbool twee regels bestaan. Dat lijkt ambiguïteit te impliceren van de grammatica, maar dat is niet zo: ook grammatica Arit2 op pagina 60 geeft aanleiding tot een niet-deterministische

PDA (met bovengaande constructie) maar Arit2 is een niet-ambigue grammatica.

Langs de andere kant kan je je afvragen of de uitspraak *indien er een deterministische PDA bestaat voor L, dan is L niet-ambigu* waar is.

De constructie van een PDA uit een CFG is zo gemakkelijk: slechts drie toestanden nodig en een heel uniforme beschrijving van de overgangen. Het valt dan ook tegen dat we omgekeerd - van PDA naar CFG - zo veel meer werk hebben ... immers, niet elke PDA heeft slechts drie toestanden en onze methode van GNFA naar GNFA met slechts twee toestanden lijkt hier niet te werken.

Voor we de constructie beschrijven, veronderstellen we eerst dat de PDA van een bepaalde vorm is:

- er is slechts één eindtoestand
- de stapel wordt leeggemaakt voor we daarin terechtkomen
- elke transitie neemt één symbool weg van de stapel, of zet er één op, maar niet beide

We hebben die vorm al eens vermeld en je kan nu een bewijsje maken dat dit niet restrictief is.

21.1 Constructie van een CFG (V, Σ, R, S) uit een PDA $(Q, \Sigma, \Gamma, \delta, q_s, \{q_f\})$:

- $V = A_{p,q}$ waarbij $p, q \in Q$
- $S = A_{q_s,q_f}$
- R bestaat uit drie delen:
  - regels van de vorm $A_{p,p} \to \epsilon$ voor elke $p \in Q$
  - regels van de vorm $A_{p,q} \to A_{p,r}A_{r,q}$ voor alle $p, q, r \in Q$
  - regels van de vorm $A_{p,q} \to aA_{r,s}b$ waarbij
    $p, q, r, s \in Q, a, b \in \Sigma_\epsilon, t \in \Gamma, (r, t) \in \delta(p, a, \epsilon), (q, \epsilon) \in \delta(s, b, t)$

De intuïtie achter de constructie is de volgende: de strings die je met een initieel lege stapel van toestand p naar q brengen met lege stapel, worden gegenereerd door het niet-eindsymbool $A_{p,q}$.

Lemma 21.3 Bovenstaande constructie van een CFG uit een PDA bewaart de taal.

Bewijs Het bewijs wordt dit jaar niet gezien. ■

Gevolg I: als een taal door een PDA wordt bepaald, dan is die taal contextvrij.

Gevolg II: elke reguliere taal is contextvrij. Dat komt doordat een FSA eigenlijk enkel maar een PDA is waarbij de stapel nooit meespeelt in de beslissingen.

Zelf doen: Er was een andere manier om in te zien dat elke reguliere taal contextvrij is: stel voor een gegeven reguliere taal een CFG op.

Afsluiter: We vermelden nog twee stellingen die inzicht geven in de structuur van contextvrije talen. De Chomsky-Schützenberger stelling zegt dat elke contextvrije taal essentieel de doorsnede is van een reguliere taal met een *geneste haakjestaal*[16] (mogelijk met meerdere soorten haakjes). De stelling van Parikh kan geformuleerd worden met behulp van een transformatie Ord van een taal: voor een gegeven string s is $Ord(s)$ de string die je verkrijgt door de symbolen van s in alfabetische volgorde te zetten. Bijvoorbeeld $Ord(bacabbc) = aabbbcc$. Parikh zegt nu dat voor elke contextvrije taal L er een reguliere taal R bestaat zodat $Ord(L) = Ord(R)$. M.a.w. afgezien van de volgorde van de symbolen zijn regulier en contextvrij gelijk!

[16] Die geneste haakjestalen worden ook wel Dyck talen genoemd, naar Walther von Dyck.

22 Een pompend lemma voor Contextvrije Talen

Stelling 22.1 Voor een contextvrije taal L bestaat een getal p (de pomplengte) zodanig dat elke string s van L met lengte minstens p kan opgedeeld worden in 5 stukken u,v,x,y en z uit Σ^* zodanig dat s = uvxyz

1. $\forall i \geq 0 : uv^ixy^iz \in L$
2. $|vy| > 0$
3. $|vxy| \leq p$

Bewijs We gebruiken weer het duivenhokprincipe, maar nu op de parsetree voor lang genoege strings. Neem eerst een CFG in Chomsky normaalvorm voor L: dat werkt iets gemakkelijker, want elke regel heeft nu ofwel twee ofwel nul niet-terminalen aan de rechterkant. Laat het aantal niet-eindsymbolen in de CFG n zijn.

Voor een string s uit L bestaat er een parse tree. Als je van die boom de onderste takken wegsnoeit hou je een volledige binaire boom over want de grammatica staat in Chomsky normaalvorm. Die boom heeft hoogte minstens gelijk aan $log_2(|s|)$. Het langste enkelvoudig pad van de wortel van die boom bevat dus minstens $log_2(|s|)+1$ knopen en als we s lang genoeg kiezen, dan is $log_2(|s|)+1$ groter dan n en bijgevolg moet er op dat langste pad minstens één niet-eindsymbool - zeg X - herhaald worden. Neem de laagste X (noteren we door X_2) en zijn dichtste herhaling (X_1) op dat pad - X is zeker verschillend van het startsymbool (waarom?). Zie figuur 2.25 voor een voorstelling van de zaken. We kunnen nu uit die parse tree een afleiding construeren waarvan we enkel wat tussenstappen laten zien:

$$S \Rightarrow^* uX_2z \Rightarrow^* uvX_1yz \Rightarrow^* uvxyz \text{ (a)}$$

In die afleiding zijn u,v,x,y,z strings uit Σ^* en bovendien zijn v en y niet tegelijkertijd leeg, want dan zou men uit X zichzelf kunnen afleiden en dat kan niet wegens de vorm van de grammatica.
Vermits (a) een geldige afleiding is, is

$$S \Rightarrow^* uX_2z \Rightarrow^* uxz$$

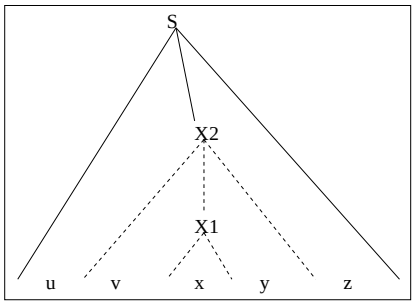
dat ook en ook

$$S \Rightarrow^* uXz \Rightarrow^* uvXyz \Rightarrow^* uvvxyyz$$

is er eentje en ...

We hebben dus al (1) en (2) van de opgave van de stelling, als we strings nemen die langer zijn dan $2^{(n-1)}$: dat wordt onze pomplengte p.

We besluiten nu ook (3): vxy wordt afgeleid vanuit X met een parse tree die kleiner is dan n, dus hoogstens 2^{n-1} bladeren heeft en die corresponderen juist met vxy. Figuur 2.25 verduidelijkt dat nog eens. ■



Figuur 2.25: De parse tree met de repeterende non-terminal X

22.1 Toepassing van het pompend lemma voor CFL's

Neem de taal $L = \{a^n b^n c^n | n \geq 0\}$. Stel dat er een pomplengte p bestaat. Neem een string s die langer is, namelijk $s = a^p b^p c^p$. Stel dat $s = uvxyz$ met $|vy| > 0$. Dan zijn er twee mogelijkheden:

1. v is van de vorm α^k en y is van de vorm β^l waarbij α en β in $\{a, b, c\}$ zitten; daarbij is $k + l > 0$; in dat geval kan uv^2xy^2z niet bestaan uit een gelijk aantal a's, b's en c's

2. v of y bevat meer dan één symbool uit $\{a, b, c\}$; in dat geval bevat v^2 of y^2 die symbolen niet in de juist volgorde en zit uv^2xy^2z niet in de taal

Gevolg: s kan niet gepompt worden en dus is L niet contextvrij.

23 Een algebra van contextvrije talen?

Voor de unie van twee contextvrije talen bepaald door de grammatica's CFG_1 en CFG_2[17] kunnen we gemakkelijk een CFG maken: stel dat de startsymbolen S_1 en S_2 zijn, maak dan een grammatica met de unie van de regels van de CFG_i, voeg de regels $S_{new} \rightarrow S_i$ toe en neem S_{new} als nieuwe startsymbool. Daarmee is bewezen dat de verzameling van contextvrije talen gesloten is voor de unie operator.

Hoe zit het met de doorsnede van CFL's? Neem als eindsymbolen $\{a, b, c\}$ en definieer $L_1 = \{a^n b^n c^m | n, m \geq 0\}$ en $L_2 = \{a^n b^m c^m | n, m \geq 0\}$. Het is duidelijk dat de L_i contextvrij zijn[18]. De doorsnede van de L_i is echter de taal $\{a^n b^n c^n | n \geq 0\}$ en daarvan hebben we in sectie 22.1 bewezen dat die taal niet contextvrij is. Dus: de doorsnede van contextvrije talen is niet noodzakelijk contextvrij.

We kunnen nu ook direct besluiten dat het complement van een CFL niet contextvrij hoeft te zijn, want $A \cap B = \overline{(\overline{A} \cup \overline{B})}$.

Voor L contextvrij en A regulier kunnen we tenslotte ook nog kijken naar de vragen:

- is $L \cup A$ contextvrij/regulier?
- is $L \cap A$ contextvrij/regulier?

Wat denk je ervan?

Zelf doen: Neem $\Sigma = \{a, b\}$. De taal $\{ss | s \in \Sigma^*\}$ is niet contextvrij. Bewijs dat. Laat zien dat het complement van die taal wordt gegenereerd door de contextvrije grammatica:

- S → AB | BA | A | B
- A → CAC | a
- B → CBC | b
- C → a | b

[17] Hernoem eerst de niet-eindsymbolen zodat ze disjunct zijn.
[18] Stel een CFG op voor die talen.

24 Ambiguïteit en determinisme

We kijken hier wat meer in detail naar het verband tussen de inherente ambiguïteit van een contextvrije taal en zijn determinisme. We noteren door DCFL de verzameling van contextvrije talen die een deterministische PDA hebben (DPDA).

Een ambigue taal kan onmogelijk deterministisch zijn, dus deterministische talen zijn niet-ambigu.

Omgekeerd is niet waar: er bestaan niet-ambigue talen die niet deterministisch zijn. Een standaard voorbeeld is de taal $\{s\hat{s}|s \in \{a,b\}^*\}$ waarin $\hat{s}$ de omgekeerde string betekent. Een niet-ambigue grammatica voor deze taal is

$$\text{S} \rightarrow \text{aSa} \mid \text{bSb} \mid \epsilon$$

maar een PDA weet van tevoren niet waar het midden van de string is en moet daarnaar *raden*: dat is de essentie van het niet-determinisme nodig om die taal te parsen. Dat betekent natuurlijk niet dat er geen deterministische parser mogelijk is voor deze taal: je kan er gemakkelijk eentje schrijven in Java. Maar het kan niet met een deterministische PDA.

Een andere manier om een niet-deterministische taal te vinden is te steunen op de eigenschap dat het complement van een DCFL ook DCFL is [19]. Neem nu de taal $L = \{ss|s \in \{a,b\}^*\}$. Je kan met het pompend lemma bewijzen dat L niet CFL is. $\overline{L}$ is echter contextvrij: pagina 75 bevat er een CFG voor. Bijgevolg is $\overline{L}$ geen DCFL.

Voorbeelden van niet-deterministische talen worden dikwijls verkregen door de unie te nemen van twee CFL's die overlappen: de taal

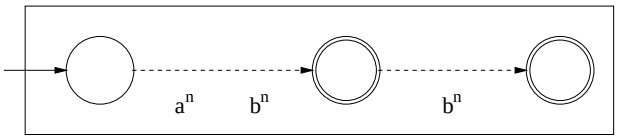
$$L = \{a^m b^n c^k | m \neq n\} \cup \{a^m b^n c^k | n \neq k\} \text{ is de unie van twee DCFL's.}$$

Stel dat L een DCFL was, dan zou zijn complement dat ook zijn, en dan ook de intersectie van dat complement met de reguliere taal $\{a^*b^*c^*\}$, maar dat geeft de taal $\{a^n b^n c^n | n \in \mathbb{N}\}$ en die taal is niet eens contextvrij!

Hier is nog een niet-deterministische taal en een schets van een constructie die dat bewijst: $L = \{a^n b^n | n \in \mathbb{N}\} \cup \{a^n b^{2n} | n \in \mathbb{N}\}$.

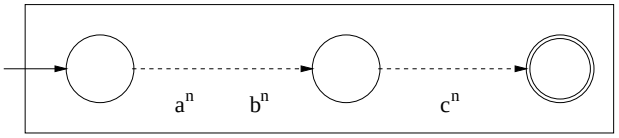
[19] Voor een bewijs zie bijvoorbeeld het boek van Kozen.

Stel dat er een DPDA M bestaat voor L, dan heeft die de structuur zoals in figuur 2.26.



Figuur 2.26: De DPDA voor L

Vervang in het laatste deel de b's door c, en maak van de linkse accepterende toestand een gewone toestand. Dan krijg je de machine in figuur 2.27.



Figuur 2.27: De DPDA voor L'

Deze PDA accepteert de taal $L' = \{a^n b^n c^n | n \in \mathbb{N}\}$ maar we weten al dat die laatste taal niet contextvrij is!

Het boek van Linz bevat een variante met meer details.

Het belang van bovenstaande redenering/voorbeeld is dat wat waar was voor reguliere talen (elke reguliere taal wordt bepaald door een deterministische FSA) niet waar is voor CFL's en PDA's: door één stapje hoger in de Chomsky hiërarchie te gaan, speelt niet-determinisme ineens een belangrijke rol.

25 Praktische parsingtechieken

Dikwijls worden we geconfronteerd met het probleem: gegeven de specificatie van een taal, maak er een herkenner voor. Die taal kan bijvoorbeeld zijn wat in een vakje op een formulier kan worden ingevuld: dat kan afhangen van wat er elders al is ingevuld (bijvoorbeeld mag je niet 3 namen van kinderen invullen, als je elders declareerde dat je er maar 2 hebt). Die taal kan een programmeertaal zijn met een ingewikkelde structuur: geneste if-then-else, statement blocks, aritmetische expressies, package qualificaties ... Meestal is zulk een taal contextvrij en het is de gewoonte om de beschrijving van een programmeertaal in twee niveaus te doen: het eerste niveau is lexicaal (zie pagina 48) en het andere syntactisch. Men maakt dan gebruik van de BNF-notatie. BNF staat voor Backus[20]-Naur[21] form en werd voor de eerste keer gebruikt om Algol[22] te beschrijven[23]. BNF trekt hard op CFG en we hebben het al min of meer gebruikt op pagina 55 voor de grammatica van Stat.

Net zoals flex vanuit een reguliere expressie omzet naar een efficiënte lexer, bestaan er tools die een CFG omzetten in een efficiënte syntax analyser. De algemene constructie van een PDA uit een CFG is niet goed genoeg o.a. omdat het bijna altijd een niet-deterministische automaat oplevert. Het deterministisch maken van een PDA is niet eenvoudig (en zelfs niet altijd mogelijk zoals we ondertussen weten). Daarom worden dikwijls extra beperkingen opgelegd aan de talen/grammatica's die zulke parsergeneratoren aankunnen.

Bekende parsergeneratoren zijn Bison (de opvolger van Yacc die C genereert) en ANTLER (Java). Het praktische gebruik van deze tools ligt buiten het bestek van deze cursus.

De automatisch geconstrueerde PDA uit een CFG gaf ons een meest-linkse top-down afleiding: we vertrekken van het startsymbool en de boom wordt vanuit de wortel opgebouwd. Dat kan verwezelijkt worden in een (deterministisch) programma door een *recursive descent parser* m.b.v. een aantal wederzijds recursieve procedures die overeenstemmen met de non-terminals in de grammatica. Dat werkt enkel als de grammatica CFG van het type *LL(k)* is. Daarin staat de eerste L voor: de input van Links naar rechts lezen; de tweede L staat voor

[20] Turing award in 1977

[21] Turing award in 2005

[22] Zoek Algol op: het is een belangrijke stap in language design geweest!

[23] Er is een hele historie aan verbonden, o.a. of Naur er wel bij moet, en of anderen niet het krediet moeten krijgen ... kijk eens in wikipedia

Leftmost derivation; de k staat voor het aantal tekens van de invoer waarop de volgende beslissing genomen wordt, of wat men de *look-ahead* noemt.

Er is ook een andere manier: we gaan over de invoerstring van links naar rechts en telkens als we iets zagen dat een rechterkant van een regel is, dan gebruiken we die regel *omgekeerd*. Hier weer het voorbeeld $a + a * a$ in een tabel zodat je kan zien wat al bekeken was en wat niet.

al gezien	nog te bekijken	te nemen actie
	a+a∗a	shift
a	+a∗a	reduce
E	+a∗a	shift
E+	a∗a	shift
E+a	∗a	reduce
E+E	∗a	reduce
E	∗a	shift
E∗	a	shift
E∗a		reduce
E∗E		reduce
E		stop

Tabel 2.4: Bottom-up parsing van $a + a * a$

De acties komen overeen met wat een shift-reduce parser doet: ofwel het eerste teken van de input op de stack zetten (en de input pointer shiften), ofwel wat op de stack staat reduceren m.b.v. een omgekeerde grammaticaregel. De basis waarop de beslissing genomen wordt is hier niet aangeduid, maar is voor shift-reduce parsers de huidige toestand (die hier helemaal niet vermeld is) en de k eerste input symbolen. Zulk een parser heet dan LR(k), waarbij de R aanduidt dat we een meest-rechtse afleiding maken.

Het construeren van praktische parsers is een uitgebreid onderzoeksdomein waarvan de verworvenheden nu gebruikt worden o.a. bij de constructie van compilers, analyse van XML documenten ... Ook het automatisch herstellen van fouten, genereren van syntax-directed editors en relevante foutenmeldingen gebeurt op basis van grammatica's.

26 Contextsensitieve Grammatica

In de stap van regulier naar contextvrij hebben we recursie over de non-terminals toegelaten, maar de linkerkant van een regel moest altijd bestaan uit juist één nonterminal.

De volgende laag in de Chomsky-hiërarchie wordt gevormd door de contextsensitieve talen, gegenereerd door contextsensitieve grammatica's: de linkerkant van een regel mag nu een bijna willekeurige string zijn van terminals en nonterminals, waarvan er eentje herschreven wordt. Bijvoorbeeld:

$$a\underline{X}Yz \rightarrow a\underline{bCD}Yz$$

is een contextsensitieve grammaticaregel: enkel in de context van a (langs links) en Yz (langs rechts) mag X herschreven worden tot bCD. Er zijn alternatieve (equivalente) definities van wat juist een contextsensitieve grammaticaregel is, maar we gaan er hier niet dieper op in.

Een equivalente definitie van een contextsensitieve grammatica is dat in een regel van de vorm $\alpha \rightarrow \beta$, altijd $|\alpha| \leq |\beta|$.

Het is a priori natuurlijk niet duidelijk dat niet alle contextsensitieve talen ook contextvrij zijn. Maar neem de taal $\{a^n b^n c^n | n \in \mathbb{N}\}$ waarvan we vroeger aantoonden dat die niet contextvrij is. Hier is een contextsensitieve grammatica (volgens de equivalente definitie) voor die taal:

S $\rightarrow$ abc

S $\rightarrow$ aSBc

cB $\rightarrow$ Bc

bB $\rightarrow$ bb

Zelf doen : bewijs dat de bovenstaande grammatica inderdaad de gevraagde taal bepaalt.

Er bestaat ook een normaal vorm voor contextsensitieve grammatica's (die de lege string niet genereren), nl. de Kuroda normaal vorm: elke regel is van de vorm

$$AB \to CD$$

$$A \to BC$$

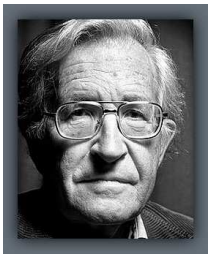
$$A \to B$$

$$A \to a$$

met A,B,C en D nonterminals, en a een terminal.

Een contextsensitieve taal kan geparsed worden door een linair begrensde automaat (LBA) - een klasse automaten die strikt sterker is dan de push down automaten, en die we in een volgend hoofdstuk zullen invoeren. Pas als we wat eigenschappen ervan kennen, zullen we zijn plaats in de hiërarchie appreciëren. LBA's zijn belangrijk omdat ze beslissingsproblemen oplossen in $O(1)$-space!

Daarmee zijn we bijna aan de top van de Chomsky hiërarchie: als we de laatste restrictie op de linkerkant van een regel wegnemen, dan komen we bij de *unrestricted grammars*. De machinerie nodig om de bijbehorende talen te parsen komt van de Turing machines: ook die worden in een volgend hoofdstuk ingevoerd.



De hiërarchie zelf vind je in tabel 2.5: er zijn verfijningen (o.a. i.v.m. determinisme) maar die hebben we niet vermeld.

	Grammatica	Taal	Automaat
Type-0	unrestricted	herkenbaar	Turingmachine
Type-1	context-sensitief	context-sensitief	lineair-begrensd
Type-2	context-vrij	context-vrij	push-down
Type-3	regulier	regulier	eindig

Tabel 2.5: De Chomsky-hiërarchie

Chomsky is ook bekend om zijn politiek activisme - de moeite om eens te lezen.

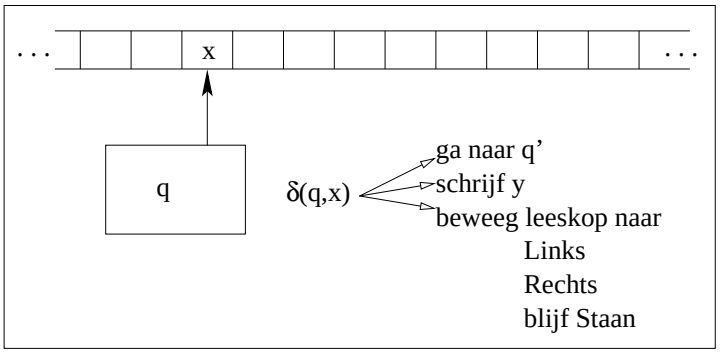
Hoofdstuk 3

Talen en Berekenbaarheid

1 De Turingmachine als herkenner en beslisser

In vorige hoofdstukken hebben we twee klassen uit de Chomsky hiërarchie van dichtbij bekeken, met hun bijbehorende herkenners. We weten al dat reguliere talen contextvrij zijn en dat er talen bestaan die niet contextvrij zijn. Het is tijd om machinerie op te zetten om zekere niet-contextvrije talen te herkennen of te beslissen. De machine die we zullen gebruiken is de Turingmachine: die is krachtiger dan de LBA die we al eens vermeld hebben. Op de LBA komen we later nog terug in dit hoofdstuk.

Ruwweg bestaat een Turingmachine uit een twee-zijdig onbegrensde band die verdeeld is in vakjes, een controle-eenheid, en een leeskop die op elk ogenblik de inhoud van een vakje op de band kan lezen. De actie van de machine hangt af van de inwendige toestand van de controle-eenheid en het teken onder de leeskop. Figuur 3.1 toont die onderdelen en ook de actie beschreven door de δ van de Turingmachine.



Figuur 3.1: Schema van een Turingmachine

Meer formeel:

Definitie 1.1 Turingmachine
Een **Turingmachine** is een 7-tuple $(Q, \Sigma, \Gamma, \delta, q_s, q_a, q_r)$ waarbij Q, Σ, Γ eindige verzamelingen zijn en

- Q is een verzameling toestanden
- Σ is een input alfabet dat # niet bevat
- Γ is het tape alfabet; $\# \in \Gamma$ en $\Sigma \subset \Gamma$
- q_s is de starttoestand
- q_a is de accepterende eindtoestand
- q_r is de verwerpende eindtoestand (r van reject) verschillend van q_a
- δ is de transitiefunctie: een totale functie met signatuur
$$Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$$

De machine wordt geïnitialiseerd als volgt:

- een eindig aantal symbolen uit het input alfabet worden aaneensluitend op de band gezet: dat is de inputstring; op de rest komt een #
- de machine wordt in toestand q_s gezet
- de leeskop wordt gepositioneerd op het meest linkse teken van de inputstring; als de string leeg is, dan op een willekeurig hekje

De machine werkt nu als volgt: op basis van het teken onder de leeskop en de toestand van de machine (dus $Q \times \Gamma$) wordt met behulp van δ bepaald naar welke toestand de machine moet overgaan, welk symbool er moet geschreven worden en welke beweging de leeskop maakt (dus $Q \times \Gamma \times \{L, R, S\}$). Dit wordt uitgevoerd en dat blijft zo doorgaan

- totdat de machine in q_a komt: de inputstring is geaccepteerd en eventueel beschouw je wat nu op de band staat als het resultaat van een berekening

- totdat de machine in q_r komt: de inputstring is verworpen
- en blijft zo doorgaan ... de machine zit in een *oneindige lus*

We hebben reeds met de Turingmachine kennisgemaakt in een vroegere cursus, dus moeten we er hier niet heel diep op ingaan. Wel willen we nog de klassieke varianten van de definitie van Turingmachine aangeven: eventueel aan jullie om te bewijzen dat die alle *equivalent* zijn (maar je zal zelf een redelijke notie van equivalentie moeten formaliseren).

- de leeskop kan enkel naar Links of Rechts, maar niet blijven Staan
- er is meer dan één accepterende toestand en/of meer dan één verwerpende toestand
- de band is onbegrensd in slechts één richting
- er is meer dan één band
- het inputalfabet heeft slechts twee symbolen
- er zijn slechts drie toestanden
- de machine heeft twee stacks i.p.v. een oneindige band

Tenslotte nog iets over het hekje #: het wordt o.a. gebruikt om aan te geven dat een vakje op de band nog niet is beschreven (al mag de Turingmachine het later wel schrijven). In veel teksten wordt het beschreven als *het blanco symbool* en er wordt het symbool $\sqcup$ voor gebruikt. We zullen soms ook verwijzen naar ons # met *het blanco symbool*, of het *hekje*.

Voor een gegeven Turingmachine TM kunnen we Σ^* verdelen in drie disjuncte stukken:

1. de strings die door de TM worden geaccepteerd: L_{TM}
2. de strings waarvoor de TM niet stopt: ∞_{TM}
3. de rest

We gebruiken die verzamelingen in de volgende definities.

Definitie 1.2 Herkennen
Een Turingmachine TM herkent L_{TM}

Een aansluitende definitie is natuurlijk

Definitie 1.3 Turing-herkenbare taal
Een taal L is Turing-herkenbaar indien er een Turingmachine TM bestaat zodanig dat $L = L_{TM}$

Laten we meteen een voorbeeld geven van een herkenbare taal L en een TM die L herkent: $\Sigma = \{a, b\}$, en $L = \{a\}^*$. De volgende tabel beschrijft δ:

Q	Γ	Q	Γ	LRS
q_s	a	q_s	#	R
q_s	b	x	b	S
q_s	#	q_a	-	-
x	a	x	a	S
x	b	x	b	S
x	#	x	#	S

Tabel 3.1: TM die $\{a\}^*$ herkent

Het is duidelijk dat ∞_{TM} niet leeg is: voor elke string niet in L gaat de machine in een lus. Voor dezelfde taal L bestaat ook een TM die altijd stopt, bijvoorbeeld:

Q	Γ	Q	Γ	LRS
q_s	a	q_s	#	R
q_s	b	q_r	-	-
q_s	#	q_a	-	-

Tabel 3.2: TM die $\{a\}^*$ beslist

Dat onderscheid vatten we in de volgende definities:

Definitie 1.4 Beslissen
Een Turingmachine TM beslist een taal L, als TM L herkent en bovendien $\infty_{TM} = \emptyset$

en

Definitie 1.5 Turing-beslisbare taal
Een taal L heet Turing-beslisbaar als er een Turingmachine is die L beslist.

A priori is het niet duidelijk dat herkennen en beslissen verschillen van elkaar, maar het zal blijken dat er een belangrijk onderscheid tussen bestaat. Wat wel duidelijk moet zijn: een beslisbare taal is ook herkenbaar.

Definitie 1.6 Co-herkenbaar/co-beslisbaar
Een taal L is **co-herkenbaar/co-beslisbaar** als $\overline{L}$ herkenbaar/beslisbaar is.

Stelling 1.7 Als L beslisbaar is, dan is L ook co-beslisbaar.

Bewijs In de Turingmachine die L beslist, verwissel je de rol van q_a en q_r. ■

Stelling 1.8 Als L herkenbaar is en co-herkenbaar, dan is L beslisbaar.

Bewijs Laat M1 de machine zijn die L herkent, en M2 de machine die $\overline{L}$ herkent. De idee is nu dat we M1 en M2 samen laten lopen als een nieuwe machine M, in parallel: zodra M1 accepteert, dan accepteert M, en zodra M2 accepteert, dan verwerpt M. M1 en M2 kunnen niet samen accepteren, en voor elke string zal minstens één van de machines M1 en M2 stoppen - in zijn aanvaardende toestand. M beslist L.

De bovenstaande constructie van M is informeel en eigenlijk moeten we die formeler maken. Dat kan door M te definiëren als een 2-tape machine: je kan zelf de details uitwerken. ■

Stelling 1.9 Er bestaat een taal die niet herkenbaar is.

Bewijs Het bewijs steunt op het begrip kardinaliteit: we weten van vroeger dat het aantal Turingmachines aftelbaar oneindig is. We weten ook dat elke Turingmachine juist één taal herkent[1]. En tenslotte weten we ook dat het aantal talen niet-aftelbaar oneindig is, want de verzameling talen is $\mathcal{P}(\Sigma^*)$. Bijgevolg bestaat een niet-herkenbare taal. In feite is daarmee zelfs bewezen dat er niet-aftelbaar veel niet-herkenbare talen bestaan. Meer nog: bijna alle talen zijn niet-herkenbaar! ■

Wat zijn we van plan met Turingmachines en die definities ...

- het onderscheid tussen beslissen en herkennen verder bestuderen
- reguliere en contextvrije talen in dit plaatje inpassen
- voorbeelden van talen bekijken die niet-herkenbaar zijn of niet-beslisbaar

De engelse termen zijn:

- herkenbaar: recognisable en recursive enumerable
- beslisbaar: decidable en recursive

In sectie 9 zullen we de historische reden voor de term *enumerable* zien.

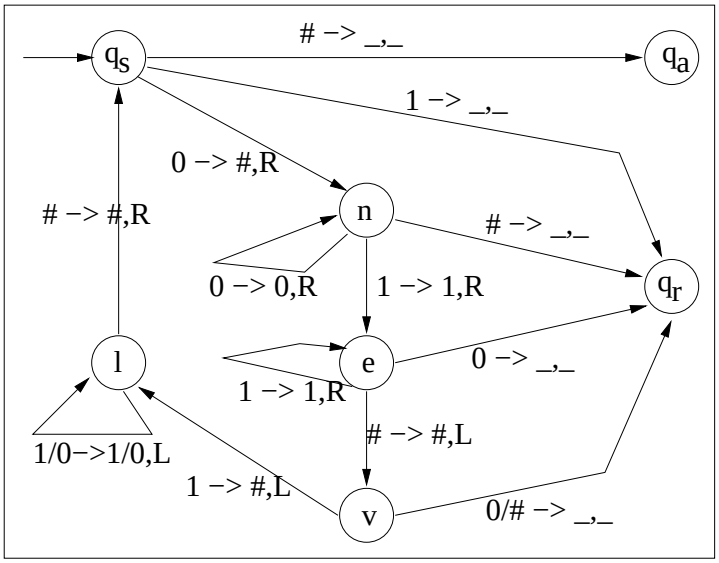
Zelf doen: Wat denk je van de uitspraken:

- elke string is herkenbaar
- elke string is beslisbaar
- elke eindige taal is beslisbaar/herkenbaar
- de unie/doorsnede van twee herkenbare talen is herkenbaar
- de doorsnede van een herkenbare en een beslisbare taal is beslisbaar

[1] Zorg dat je snapt waarom!

2 Grafische voorstelling van een Turingmachine

Net zoals bij FSA's en PDA's kunnen we een Turingmachine goed visualiseren met een graaf: daarin zijn de knopen de toestanden van de TM. Een voorbeeld verduidelijkt veel: figuur 3.2 toont de voorstelling met een graaf van een TM die (hopelijk) de taal $\{0^n 1^n | n \geq 0\}$ beslist.



Figuur 3.2: Grafische voorstelling van een Turingmachine

Vanuit elke toestand vertrekken zoveel bogen als er elementen zijn in Γ (maar sommige worden samengenomen). Het label op een boog symboliseert de δ van de machine. Een _ wordt gebruikt om aan te duiden dat het er niet toe doet welk symbool er staat: typisch wanneer het een overgang naar een eindtoestand betreft.

We kunnen in woorden beschrijven wat die machine doet met een input van nullen en enen (en ook de lege string).

- als in de starttoestand een
  - # gezien wordt, dan accepteer
  - 1 gezien wordt, dan verwerp
  - 0 gezien wordt, veeg die uit en ga rechts naar de toestand n die alle nullen overslaat
- als in toestand n een
  - # gezien wordt, verwerp
  - 1 gezien wordt, ga rechts naar e die alle enen overslaat
  - 0 gezien wordt, ga naar rechts
- in toestand e
  - zolang 1-en gezien worden, ga naar rechts
  - bij een 0, verwerp
  - bij een #, ga links naar toestand v die één 1 zal uitvegen
- in v
  - een 0 of # zijn fout, dus verwerp
  - veeg de 1 uit en ga links naar toestand l die de linkerkant van de string zal opzoeken
- in toestand l
  - ga naar links zolang je 0 of 1 ziet
  - bij #, ga rechts naar de begintoestand

Zoals je ziet heeft elke toestand zijn eigen functie.

De TM in figuur 3.2 toont aan dat de taal $\{0^n1^n|n \geq 0\}$ beslisbaar is.

3 Berekeningen van een TM voorstellen en nabootsen

We willen dikwijls een *configuratie* van een TM compact voorstellen: een configuratie is dan de toestand van de TM, de band en waar de leeskop zich bevindt. Vermits de band op elk ogenblik tijdens een uitvoering slechts een eindig aantal tekens verschillend van # bevat - of alternatief: er is altijd een linker- en rechteroneindig stuk van de band met enkel # - kunnen we de volgende notatie gebruiken:

$\alpha q \beta$

stelt voor dat de band $\alpha\beta$ bevat, en links en rechts ervan enkel #; de machine is in toestand q en de leeskop van de machine bevindt zich op het eerste teken van β. α en β mogen # bevatten.

Een beginconfiguratie is nu altijd van de vorm $q_s\alpha$

Een eindconfiguratie is van de vorm $\alpha q_a \beta$ (een accepterende configuratie) of $\alpha q_r \beta$ (een verwerpende configuratie).

Tijdens de uitvoering van een TM zijn twee opeenvolgende configuraties verbonden door δ op de volgende manier:

$\alpha qb\beta \rightarrow \alpha pc\beta$ indien $\delta(q,b) = (p,c,S)$
$\alpha aqb\beta \rightarrow \alpha pac\beta$ indien $\delta(q,b) = (p,c,L)$
$\alpha qb\beta \rightarrow \alpha cp\beta$ indien $\delta(q,b) = (p,c,R)$

Die overgangen moet je nog aanvullen voor het geval dat α of β leeg zijn: dan moet je een extra # erbij zetten.

Een opeenvolging van configuraties van een beginconfiguratie tot een eindconfiguratie noemt men een *computation history*.

Het is nu redelijk gemakkelijk om in te zien dat de berekeningen van een willekeurige TM door een programmeertaal zoals Prolog of Java kunnen gesimuleerd worden. Om dat wat concreter te maken geven we gewoon wat Prologcode. Een configuratie $\alpha q \beta$ wordt voorgesteld door een term met hoofdfunctor conf/3 als volgt:

$conf([a_n, ..., a_2, a_1], q, [b_1, b_2, ..., b_m])$, met $a_1a_2...a_n = \alpha$ en $b_1b_2...b_m = \beta$ waarbij de a_i en b_j in Γ zitten.[2]

[2]Zoek eens op *Huet Zipper*

Een deel van het Prologprogramma dat op configuraties werkt:

```
onestep(conf(Alpha,Q,[]), NewConf) :-
          onestep(conf(Alpha,Q,[#]), NewConf).
onestep(conf(Alpha,Q,[B|Beta]), conf(Alpha,P,[C|Beta])) :-
          delta(Q,B,P,C,stop).
```

Zelf doen: Breid bovenstaand programma uit, en leg in woorden uit wat de functie is van elke clause.

Omgekeerd moeten we kunnen inzien dat met een Turingmachine ook een Prologprogramma kan worden gesimuleerd: om dat "direct" te doen vraagt veel werk. Hier is een klassieke omweg: eerst toon je dat Prolog kan geïmplementeerd worden op een Intel machine; dat is niet moeilijk: SWI-Prolog is een voorbeeld; dan toon je aan dat de Intel machine kan gesimuleerd worden met een register machine; een register machine is al een theoretische constructie en van daar naar de Turingmachine is nog maar een kleine stap.

Daarmee is de kring rond. Als een Turingmachine elk X-programma kan simuleren, en een X-programma elke Turingmachine, dan hebben X en TM's dezelfde berekeningskracht. Pas op: voor ons betekent dat niets i.v.m. complexiteit, enkel i.v.m. welke talen kunnen beslist/herkend worden.

In de toekomst zullen we daarvan meer dan eens gebruik kunnen maken.

Zelf doen: Turingmachines samenstellen
We hebben dikwijls nodig dat een aantal Turingmachines $TM_1, TM_2 ... TM_n$ gecombineerd worden tot een nieuwe Turingmachine TM. We gebruiken woorden zoals

TM roept TM_1 op als een subroutine

en

als TM_1 stopt in een aanvaardende eindtoestand, dan laat TM TM_2 lopen op de string s

Zorg dat je weet wat zulke uitspraken willen zeggen, t.t.z. formaliseer die tot op zekere hoogte.

4 Niet-deterministische Turingmachines

We vermelden niet-deterministische Turingmachines hier enkel omdat elk boek over berekenbaarheid het doet: we zullen er verder geen gebruik van maken in de studie van berekenbaarheid. Niet-deterministische Turingmachines zijn wel belangrijk in de context van complexiteit.

In de definitie van niet-deterministische Turingmachine heeft δ signatuur $Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R, S\})$, t.t.z. dat er vanuit een gegeven configuratie mogelijk meer dan één configuratie kan bereikt worden. Een string wordt door de machine geaccepteerd als er een computation history bestaat die eindigt met een accepterende configuratie.

Een niet-deterministische Turingmachine kan door een deterministische gesimuleerd worden. Sommige boeken beschrijven een simulatie van een NDTM door een DTM. Voor ons is het gemakkelijker te zeggen: het Prologprogramma dat we vroeger (gedeeltelijk) gaven, kunnen we met een meta-vertolker uitvoeren die werkt volgens iterative deepening (of breedte-eerst) en die laten stoppen zogauw we in de accept toestand komen[3]. Als we in Prolog een NDTM kunnen simuleren, dan dus ook met een DTM.

5 Talen, $\mathcal{P}(\mathbb{N})$, eigenschappen en terminologie

Wij hebben herkenbaarheid/beslisbaarheid geformuleerd in termen van talen over een alfabet. Even gebruikelijk is het om die concepten te definiëren in de context van deelverzamelingen van $\mathbb{N}$. Die twee zijn equivalent omdat we $\mathbb{N}$ kunnen beschouwen als $\{0,1\}^*$, dus alle strings over het alfabet $\{0,1\}$ (binaire notatie, maar een andere zou ook goed zijn), en een deel van $\mathbb{N}$ is dus een taal over dat alfabet. Soms reserveert men de terminologie *(recursive) enumerable* voor zulke verzamelingen (of talen) en de termen *decidable* en *recognisable* voor eigenschappen. Dat onderscheid is niet echt belangrijk: stel dat E een eigenschap is die elementen van een verzameling V kunnen hebben of niet, dan kan je definiëren: $\{x | x \in V, x\ heeft\ eigenschap\ E\}$. Dan zie je duidelijk dat eigenschappen en subsets een gelijkwaardig zicht op de problematiek geven.

Het woord *recursive* roept bij ons meestal het beeld op van een functie (methode, predicaat, procedure ...) die zichzelf oproept, of een data type dat in termen van

[3] Niet de meest efficiënte manier, maar die analyse herinner je je nog van FVI natuurlijk.

zichzelf is gedefinieerd (lijst, boom, ...). In de term *recursive enumerable* heeft het stukje *recursive* daar niet rechtsreeks mee te maken.

6 Encodering

Als we informeel een taal beschrijven, dan moeten we niet denken aan de encodering ervan. Zo kunnen we spreken over *alle vlakke grafen* maar geen specifiek alfabet in gedachten hebben waarin we die verzameling als taal kunnen beschrijven. Maar als we een algoritme willen schrijven om te beslissen of een graaf vlak is, dan is een alfabet nodig en een mapping van de grafen naar strings. Zulk een mapping is een encodering. Encoderingen moeten *redelijk* zijn. De volgende eigenschappen moeten minstens vervuld zijn (als voorbeeld i.v.m. grafen):

- elke graaf encodeert naar één string
- twee *verschillende* grafen encoderen naar twee verschillende strings
- van een string moet beslist kunnen worden of ie een graaf encodeert en welke graaf

Een redelijke encodering introduceert ook geen extra informatie.

Wat voorbeelden:

1. Je wil de priemgetallen bekijken als verzameling. Je kiest voor een unaire representatie van getallen; het alfabet is {@}. Dus 7 wordt voorgesteld door 7 keer het symbool @: @@@@@@@

   Dit is redelijk.

2. Je wil de priemgetallen bekijken als verzameling. Je kiest voor een binaire representatie van getallen; het alfabet is {@,!}. Dus 9 wordt voorgesteld door !@@!

   Dit is redelijk.

3. Je wil de priemgetallen bekijken als verzameling. Je kiest voor een representatie van getallen in twee delen: het eerste deel is een bit die aanduidt of het getal priem is of niet (voorgesteld door p en n); het tweede deel is een unaire representatie van het getal met alfabet {@}. Dus 7 wordt voorgesteld door p@@@@@@@ en 9 door n@@@@@@@@@

   Dit is niet redelijk: je encodering heeft extra informatie.

Misschien heb je de indruk dat die eis je tegenwerkt. Immers, met die laatste representatie is het beslisbaar in constante tijd of een getal priem is of niet, want je hoeft alleen maar naar het eerste symbool te kijken. Mis! Wat gebeurt er met de string p@@@@@? Die behoort tot Σ^*, maar stelt geen getal voor, want het eerste symbool beweert dat de string een priemgetal voorstelt, terwijl het tweede deel het getal 4 voorstelt. Het kost nu evenveel werk om van een string in Σ^* te beslissen of ie een getal voorstelt als om te beslissen dat een getal priem is in een redelijke encodering.

Vanuit het standpunt van berekenbaarheid zijn de encoderingen in (1) en (2) hiervoor equivalent: er bestaat een algoritme dat de ene in de andere omzet; je kan het met een Turingmachine implementeren; de mapping van de ene naar de andere is Turing-berekenbaar.

Vanuit het standpunt van de complexiteit zijn de encoderingen in (1) en (2) hiervoor niet equivalent! Bijvoorbeeld: twee getallen optellen in unaire notatie is O(n) waarbij n het grootste van de twee getallen is. In binaire notatie is dat O(log(n)), want het is lineair in het aantal bits nodig om de getallen voor te stellen.

De encodering van een object S zullen we aanduiden met $< S >$. Als we een encodering willen van een aantal objecten S_i, dan wordt dat gewoon $< S_1, S_2, ... >$.

Zelf doen:

Zoek redelijke encoderingen voor bomen, XML-documenten, ...

Zoek er ook wat onredelijke, eventueel voor andere objecten.

Als je een functie (bijvoorbeeld van $\mathbb{N}$ naar $\mathbb{N}$) implementeert, dan kan de output ook geëncodeerd zijn. Je kreeg als opdracht een functie te implementeren die bij input i het i-de priemgetal teruggeeft. Je was vlug klaar, want je schreef bij input i gewoon het getal i uit, met als argument dat i in de output gewoon de encodering is van het i-de priemgetal. Je professor was niet tevreden. Was dat redelijk?

7 Universele Turingmachines

Een TM kunnen we helemaal beschrijven door zijn transitietabel te geven: we kunnen die transitietabel encoderen en op een band plaatsen. We spreken daarbij bijvoorbeeld af dat we toestanden en inputsymbolen encoderen door *getallen* die we encoderen met één willekeurig symbool in unaire notatie. We hebben ook nog een nieuw symbool nodig als delimiter, om stukjes van de encodering af te scheiden van elkaar, en we vermelden apart welke de toestanden q_s, q_a en q_r van de TM zijn. De encodering van de TM hebben we op een band gezet: de programmaband; op een tweede band zetten we een inputstring (in dezelfde encodering) voor de TM: het werkgeheugen.

Nu maken we één nieuwe machine UTM met een Γ' die bovenstaande Γ bevat, en ook de delimiter en misschien ook nog andere hulpsymbolen. UTM gebruikt de twee banden als input. Als je wil mag die UTM nog extra banden hebben: we weten dat we niks essentieels toevoegen aan de kracht van Turingmachines. UTM gebruikt het programma op de programmaband om de acties van TM uit te voeren op de geheugenband. Zogauw TM zou stoppen in q_a of q_r, gaat UTM naar zijn eigen q'_a of q'_r.

Bovenstaand scenario is in meer detail te verwezenlijken, maar voor ons is dit voldoende. De Universele Turingmachine kan elke TM simuleren - als die maar geëncodeerd wordt zoals nodig: we kunnen de UTM zelfs laten beginnen met controleren of wat op de banden staat wel een geldige encodering is.

Laat je niet in de war brengen door de sectie over samenstellen van Turingmachines, waar een TM_1 een andere TM_2 als subroutine gebruikt: in dat geval zijn de toestanden van TM_2 een subset van de toestanden van TM_1. Bij de UTM is dat niet zo: UTM heeft een vast aantal toestanden, onafhankelijk van welke TM er gesimuleerd wordt; en de geëncodeerde toestanden van de gesimuleerde machine, zijn geen toestanden van de UTM.

Hoe *groot* is een UTM? Claude Shannon liet zien dat we in een TM altijd toestanden voor symbolen kunnen ruilen - zie de opgave op pagina 84. Daarom is het de gewoonte geworden om de grootte van een TM uit te drukken als $(|Q|, |\Gamma|)$, of het product $|Q| \times |\Gamma|$. Marvin Minsky vond in 1956 een (7,4) UTM. Later vond men kleinere UTM's. Van een bepaalde (2,3) machine is ondertussen bewezen dat ie universeel is - wel wat controversieel overigens. Een (2,2) TM kan niet universeel zijn.

8 Het Halting-probleem

Informeel gaat het om het volgende: gegeven een Turingmachine M en een string s, kan je bepalen of M bij input s stopt. De vraag is hier niet of M s accepteert of verwerpt: in beide gevallen stopt de machine. De vraag is of M in de derde mogelijkheid terecht komt, namelijk in een lus. Verder willen we die vraag laten beantwoorden door een algoritme, dus we willen eigenlijk een Turingmachine H die de taal $\{< M, s > | M \textit{ is een Turingmachine die stopt bij input } s\}$ **beslist**. De taal die we willen beslissen noteren we door H_{TM}.

We zullen laten zien dat H_{TM} niet beslisbaar is. We wisten al dat er niet-beslisbare talen moeten bestaan, maar H_{TM} is ons eerste concrete voorbeeld. Een gerelateerd probleem is het acceptatieprobleem voor Turingmachines. De geassocieerde taal is

$$A_{TM} = \{< M, s > | M \textit{ is een Turingmachine en } s \in L_M\}$$

A_{TM} is de eerste taal waarvoor we bewijzen dat ie niet beslisbaar is.

Stelling 8.1 A_{TM} is niet beslisbaar.

Bewijs We gebruiken *bewijs door contradictie.*

Stel er bestaat een beslisser B voor A_{TM}. Dat betekent dat bij input $< M, s >$ B accepteert als M bij input s stopt in zijn q_a en verwerpt als M bij input s stopt in zijn q_r of loopt. We schrijven

$B(< M, s >)$ is accept als M s accepteert en anders reject

Construeer nu de contradictie machine C met eigenschap:

$C(< M >) = opposite(B(< M, M >))$ voor elke Turingmachine M.

Daarbij is $opposite(accept) = reject$ en $opposite(reject) = accept$.

Neem nu voor M hierboven C zelf, dan krijgen we:

$C(< C >) = opposite(B(< C, C >))$

Als $C(< C >) = accept$, dan is $B(< C, C >) = accept$, dan is $opposite(B(< C, C >)) = reject$, dan is $C(< C >) = reject$, dan is $B(< C, C >) = reject$, dan is $opposite(B(< C, C >)) = accept$, dan is $C(< C >) = accept$ ...

Dus C kan niet bestaan, dus B bestaat niet. Dus A_{TM} is niet beslisbaar. ■

Stelling 8.2 H_{TM} is niet beslisbaar.

Bewijs Stel dat H_{TM} beslisbaar is door een Turingmachine H. We construeren nu beslisser B voor A_{TM} als volgt: bij input $< M, s >$ doet B:

- laat eerst H lopen op $< M, s >$
- als $H(< M, s >) = accept$, dan laat B M lopen op s en geeft als resultaat wat M geeft
- als $H(< M, s >) = reject$ dan reject B ook de string $< M, s >$.

Vermits er geen beslisser voor A_{TM} bestaat, kan H niet bestaan en is dus ook H_{TM} niet beslisbaar. ■

Stelling 8.3 H_{TM} is herkenbaar.

Bewijs De herkenner H voor H_{TM} laat gewoon bij input $< M, s >$ M lopen op s: als die stopt, dan accepteert H zijn input. Als M niet stopt, dan blijft H lopen op zijn input. ■

Stelling 8.4 A_{TM} is herkenbaar.

Bewijs Gelijkaardig aan de herkenner voor H_{TM}. ■

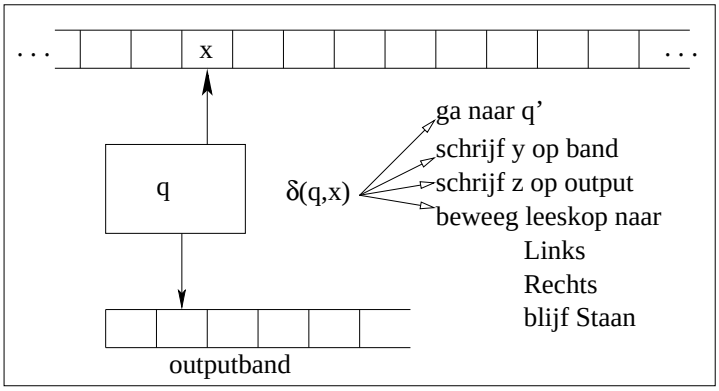
We hebben nu als direct gevolg van deze stellingen:

Gevolg 8.5 $\overline{A_{TM}}$ en $\overline{H_{TM}}$ zijn niet herkenbaar.

Bewijs Als $\overline{A_{TM}}$ herkenbaar is, en vermits A_{TM} herkenbaar is, moet A_{TM} ook beslisbaar zijn (zie stelling op pagina 86). Maar A_{TM} is niet beslisbaar, dus is $\overline{A_{TM}}$ niet herkenbaar. Idem voor H_{TM}. ■

9 De enumeratormachine

De enumeratormachine is eigenlijk de machine zoals orgineel voorgesteld door Alan Turing in zijn publicatie in 1936: hij was geïnteresseerd in het genereren van decimale expansies van *berekenbare reële getallen*. Het verband met herkenbare talen is echter direct, maar we moesten wachten met de enumerator tot na het Halting-probleem want dat zullen we gebruiken.



Figuur 3.3: Schema van een enumeratormachine

Zoals je ziet op figuur 3.3 trekt een enumerator op een Turingmachine en heeft als extra's

- een enumeratortoestand q_e
- een outputband
- een outputmarker

De δ van de enumerator heeft als signatuur $Q \times \Gamma \to Q \times \Gamma \times \Gamma_\epsilon \times \{L, R, S\}$. Daarin is de laatste Γ_ϵ bedoeld als symbool dat (bij die overgang) geschreven wordt op de outputband.

De machine start met lege band en lege outputband, in de gewone q_s en begint te werken. Telkens bij een overgang iets op de output wordt geschreven verschuift de schrijfkop naar rechts. Telkens de machine in toestand q_e komt, wordt op de output de outputmarker geschreven, overgegaan naar q_s en de machine loopt verder.

Het is mogelijk dat de enumerator niet stopt en altijd maar weer strings (gescheiden door een outputmarker) output. Het is mogelijk dat de enumerator niet stopt en blijft werken aan dezelfde outputstring (misschien zelfs de eerste!). Het is ook mogelijk dat de machine stopt en dat er een eindig aantal strings (gescheiden door de marker) op de output staan. Gelijk hoe, het heeft zin om te spreken over de verzameling (eindige) outputstrings door de enumerator geproduceerd - of geënumereerd. Die verzameling is de taal door de enumerator bepaald of geënumereerd. De enumerator mag daarbij dezelfde string meer dan eens op de output zetten. We kunnen nu bewijzen:

Stelling 9.1 De taal door een enumerator bepaald is herkenbaar en elke herkenbare taal wordt door een enumerator geënumereerd.

Bewijs (1) We beschrijven informeel een herkenner TM voor de taal L bepaald door een gegeven enumerator Enu: TM gebruikt Enu als subroutine als volgt.

> Geef een string s aan de TM. De TM start de Enu. Telkens de Enu in zijn q_e komt, kijkt TM na of de laatst geproduceerde string op de outputband van de Enu gelijk is aan s. Indien ja: TM accepteert. Indien niet, laat de Enu verderrekenen.

(2) Laat de TM L bepalen. We construeren de Enu voor die L als volgt. Maak eerst een TM_{gen} die gegeven een getal n de eerste n strings uit Σ^* op de band zet: $s_1, s_2, ...s_n$. Maak een TM_n die op elk van die n strings, n stappen van TM uitvoert: als daarbij een string s_i geaccepteerd wordt, schrijf die dan op de outputband voor de Enu. Maak nu een TM_{driver} die de opeenvolgende getallen n genereert en dan TM_{gen} en TM_n oproept.

■

Waarvoor hadden we nu het Haltingprobleem nodig? We zouden naïef de volgende procedure hebben kunnen voorstellen om een enumerator Enu te maken van een Turingmachine TM:

> genereer de strings van Σ^* in een bepaalde volgorde $s_1, s_2...$
>
> geef elke s_i als input aan TM en als TM accepteert, output s_i

Dit werkt niet omdat TM voor sommige s_i misschien niet stopt en we - dankzij het Halting-probleem - weten dat er geen manier is om dat van te voren te weten. Als dan s_{i+1} wel tot L_M behoort, dan enumereert Enu die string niet.

10 Beslisbare talen

10.1 In verband met de reguliere talen

Zomaar zeggen *een reguliere taal is beslisbaar*, is voor een aantal interpretaties vatbaar: we moeten precies zeggen welke input we aan de beslisser willen geven. Daarom zijn er van voorgaande uitspraak verschillende versies, afhankelijk van de manier waarop de reguliere taal gegeven is.

We definiëren een aantal talen:

- $A_{DFA} = \{< D, s > | D \textit{ is een } DFA, \textit{ en } D \textit{ accepteert } s\}$
- $A_{NFA} = \{< N, s > | N \textit{ is een } NFA, \textit{ en } N \textit{ accepteert } s\}$
- $A_{RegExp} = \{< RE, s > | RE \textit{ is een reguliere expressie, en } RE \textit{ genereert } s\}$

Stelling 10.1 A_{DFA}, A_{NFA} en A_{RegExp} zijn beslisbaar.

Constructief Bewijs

- De beslisser B krijgt als input $< D, s >$. B simuleert D op s. Als D s accepteert, dan stopt B in zijn q_a. Als D s verwerpt, dan stopt B in zijn q_r. Er is geen probleem met niet stoppen.
- Bij het simuleren van een NFA kan het zijn dat we in een lus gaan, dus doen we het wat anders: de beslisser B krijgt als input $< N, s >$. De NFA wordt eerst omgezet naar een DFA met het algoritme dat we zagen in sectie 11. Dan passen we de simulatie toe.
- Voor input $< RE, s >$ construeren we eerst vanuit de RE een DFA en dan passen we hierboven toe.

■

We kunnen nu ook besluiten dat een reguliere taal beslisbaar is. Probeer het subtiele verschil met bovenstaande stelling toch te waarderen (er staat een hint bij het bewijs).

Drie populaire vragen i.v.m. talen zijn de volgende:

- bevat de taal de lege string?
- is de taal leeg?
- zijn twee gegeven talen gelijk?

We kunnen voor elk ervan een beslisser maken, maar we moeten natuurlijk weer heel goed de input voor die beslisser beschrijven en dan de beslisser construeren. I.p.v. een TM te maken als beslisser, zullen we naar goede gewoonte informeel beschrijven wat ie doet.

De eerste vraag is voor reguliere talen triviaal, want A_{DFA} is beslisbaar.

Stelling 10.2 $E_{DFA} = \{< DFA > | L_{DFA} = \phi\}$ is beslisbaar.

Bewijs Er zijn veel manieren om dit te bewijzen - hier ééntje die gebruik maakt van een boel theorie (en die eigenlijk een overkill is).

Transformeer de DFA naar een minimale DFA_{min} die dezelfde taal accepteert: als $L_{DFA} = \phi$, dan is DFA_{min} isomorf met ... (teken zelf eens die machine). Dat beslissen is eenvoudig. ■

Stelling 10.3 $EQ_{DFA} = \{< DFA_1, DFA_2 > | L_{DFA_1} = L_{DFA_2}\}$ is beslisbaar.

Bewijs Er zijn weer veel manieren om dit te bewijzen - hier ééntje die gebruik maakt van de algebraïsche eigenschappen van de DFA's.

Uit DFA_1 en DFA_2 construeer je de DFA_Δ die het symmetrisch verschil tussen L_{DFA_1} en L_{DFA_2} bepaalt. Beslis dan of DFA_Δ de lege taal bepaalt m.b.v. vorige stelling. ■

Zelf doen: zoek andere bewijzen voor de stellingen hierboven.

10.2 In verband met de contextvrije talen

De vragen zijn analoog aan die bij reguliere talen en ook hier kunnen we een CFL geven door zijn CFG of door zijn PDA. We doen het enkel vertrekkend van de grammatica.

Stelling 10.4 $A_{CFG} = \{< G, s > | G \text{ is een } CFG, \text{ en } s \in L_G\}$ is beslisbaar: *acceptance* van een string door een CFG is beslisbaar.

Bewijs Een naief bewijsidee zou zijn om de CFG te gebruiken om strings te genereren en als s gegenereerd wordt te accepteren. Dat zou ons enkel een herkenner geven en geen beslisser. We maken gebruik van de Chomsky Normaal Vorm van een CFG, want die garandeert dat een string met lengte $n > 0$ enkel kan afgeleid worden in $2n - 1$ stappen. Dus:

Bij input $< G, s >$, converteer G eerst naar zijn Chomsky Normaal Vorm. Genereer alle mogelijke strings met een derivatielengte $2|s| - 1$: dat zijn er eindig veel. Indien s ertussen zit, accept, anders reject. ■

Stelling 10.5 $E_{CFG} = \{< G > | G \text{ is een } CFG, \text{ en } L_G = \phi\}$ is beslisbaar: *emptyness* van een CFL is beslisbaar.

Bewijs We beschrijven informeel een algoritme dat G transformeert naar een vorm waarin de beslissing gemakkelijk is.

- als er een regel $A \to \alpha$ in zit, en α bestaat alleen uit eindsymbolen (mag dus ook ϵ zijn), dan
  - verwijder alle regels waar A aan de linkerkant staat
  - vervang in elke regel waar A rechts voorkomt, de voorkomens van A door α
- blijf dit doen totdat ofwel
  - het startsymbool verwijderd is: reject, want het startsymbool kan een string afleiden

– er geen regels zijn van de benodigde vorm: accept, want de taal is leeg

■

Pas toch een beetje op met bovenstaand bewijs: de grammatica wordt getransformeerd naar een niet-equivalente!

Zelf doen: Geef een grammatica die de lege taal bepaalt en pas er bovenstaand algoritme op toe. Leerde je daardoor iets over Prologprogramma's die geen antwoorden (kunnen) geven?

Stelling 10.6 $ES_{CFG} = \{< G > | G \textit{ is een } CFG, \textit{ en } \epsilon \in L_G\}$ is beslisbaar: het is beslisbaar of een CFG de lege string genereert.

Bewijs We transformeren de CFG naar zijn Chomsky Normaal Vorm. Indien die de regel $S \rightarrow \epsilon$ bevat, accepteer, anders reject. ■

Stelling 10.7 Elke CFL is beslisbaar. De CFL wordt gegeven door zijn CFG.

Bewijs Hier wordt gevraagd om voor een gegeven CFG G te bewijzen dat er een beslisser B_G bestaat die voor elke string s kan beslissen of ie door G wordt gegenereerd. Dat is niet hetzelfde als A_{CFG} beslissen. Maar het trekt er voldoende op: werk de details zelf uit. ■

Er blijft nog over: is EQ_{CFG} beslisbaar, t.t.z. als we twee contextvrije grammatica's krijgen, kunnen we dan beslissen of ze dezelfde taal bepalen? De oplossing voor EQ_{DFA} steunde op het symmetrisch verschil van reguliere talen. Datzelfde idee kunnen we niet gebruiken voor CFL's want die zijn niet gesloten onder complement of doorsnede.

Zelf doen:

Bewijs dat $\overline{EQ_{CFG}}$ herkenbaar is.

Wat betekent dat voor EQ_{CFG}?

Bewijs dat Stelling 10.6 rechtsreeks volgt uit Stelling 10.4.

11 Niet-beslisbare talen

We hebben in sectie 8 bewezen dat A_{TM} en H_{TM} niet beslisbaar zijn. We doen dat hier voor nog meer talen.

Stelling 11.1 $E_{TM} = \{< M > | M \text{ is een } TM, \text{ en } L_M = \phi\}$ is niet beslisbaar: het is niet beslisbaar of een Turingmachine geen enkele input accepteert.

Bewijs Stel dat E_{TM} beslisbaar is, dan bestaat er een beslisser E voor die taal. Construeer nu een beslisser B voor A_{TM} als volgt: B krijgt als input $< M, s >$

- construeer een machine M_s die als volgt te werk gaat: voor input w doet M_s
  - indien $w \neq s$, reject
  - laat M lopen op w en geef hetzelfde resultaat terug
- laat E lopen op $< M_s >$
  - als $E < M_s >$ accepteert, dan laten we B zijn input $< M, s >$ verwerpen; immers, M_s bepaalt de lege taal, dus M accepteerde s niet
  - als $E < M_s >$ verwerpt, dan laten we B zijn input accepteren; vermits M_s niet leeg is, accepteert M s

Dit toont aan dat B een beslisser is voor A_{TM}, wat niet kan, dus bestaat E niet en E_{TM} is niet beslisbaar. ■

We kunnen E_{TM} lichtjes anders formuleren, als een instantie van het meer algemeen probleem: bepaalt de TM een taal die komt uit een gegeven verzameling van talen, ofwel

$$IsIn_{TM,S} = \{< M, S > | L_M \in S\}.$$

Dan is $E_{TM} = IsIn_{TM,\{\phi\}}$

Hier is een andere instantie van het algemene probleem:

> bepaalt een gegeven Turingmachine een reguliere taal; of is $IsIn_{TM,RegLan}$ beslisbaar; dit probleem wordt ook aangeduid met $REGULAR_{TM}$

Stelling 11.2 $REGULAR_{TM}$ is niet beslisbaar.

Bewijs Stel dat Turingmachine R een beslisser is voor $REGULAR_{TM}$. We nemen eerst twee willekeurige symbolen die we aanduiden met 0 en 1, en we maken een beslisser B voor A_{TM} als volgt: bij input $< M, s >$ doet B

- construeert een hulpmachine H_s die op input x het volgende doet:
  - als x van de vorm 0^n1^n is dan accepteer
  - anders: laat M lopen op s; als M s accepteert, dan accept
- laat R lopen op $< H_s >$
- als R accepteert, accept; als R verwerpt, reject

Bemerk eerst dat de H_s machine nooit moet lopen: ze dient alleen als input voor R. Om de redenering nu af te maken:

H_s accepteert ofwel de taal $\{0^n1^n\}$ (niet-regulier) ofwel heel Σ^* (regulier). Dus B accepteert $< M, s >$ alss $R < H_s >$ accepteert, alss H_s heel Σ^* accepteert, alss M s accepteert. Dus is B een beslisser voor A_{TM}, hetgeen niet kan, dus R bestaat niet en $REGULAR_{TM}$ is niet beslisbaar. ■

Stelling 11.3 EQ_{TM} is niet beslisbaar.

Bewijs We weten al dat E_{TM} niet beslisbaar is, en eigenlijk is dit hier een speciaal geval: het is duidelijk dat als we van twee willekeurige machines kunnen beslissen of ze dezelfde taal genereren, dan moet dat ook kunnen met een willekeurige machine en M_ϕ (de machine waarvan de taal leeg is). Bijgevolg krijgen we een contradictie door te veronderstellen dat EQ_{TM} beslisbaar is. ■

De bewijstechniek hierboven wordt opnieuw bekeken in sectie 16.

Vóór we de volgende stelling bewijzen, moeten we de klasse van *Lineair Begrensde Automaten* invoeren.

Definitie 11.4 Lineair Begrensde Automaat
Een Lineair Begrensde Automaat is een Turingmachine die niet leest of schrijft buiten het deel van de band dat initieel invoer bevat.

De naam van die automaat is misschien raar, maar een equivalente definitie laat toe dat het stuk band dat de machine gebruikt slechts een constante factor f groter mag zijn dan de invoer: die f is onafhankelijk van de grootte van de invoer natuurlijk. We hebben vroeger al vermeld dat zulke automaten context-sensitieve talen kunnen parsen.

Een beslisser om uit te maken of een gegeven Turingmachine eigenlijk een LBA is, kan niet (je kan het bewijs over $REGULAR_{TM}$ aanpassen). Maar je kan van elke Turingmachine gemakkelijk een LBA maken door links en rechts van de invoer een marker te zetten en de transitietabel van de TM aan te passen zodat die nooit overschreden wordt.[4]

Het acceptance probleem voor LBA is gedefinieerd als de taal
$$A_{LBA} = \{< M, s > | M \textit{ is een LBA en } s \in L_{LBA}\}.$$
Misschien verrassend, maar ...

Stelling 11.5 A_{LBA} is beslisbaar

Bewijs We kijken naar de configuraties die kunnen voorkomen tijdens de uitvoering van een LBA op een string met lengte n. Het aantal toestanden van de LBA noteren we met q en het aantal elementen in het bandalfabet met b. Het aantal mogelijke strings die tijdens de uitvoering op de band kunnen staan is begrensd door b^n. De leeskop kan onder elk van de symbolen staan terwijl de machine in elk van de toestanden kan zitten. Dat geeft in het totaal maximaal qnb^n configuraties.

We kunnen nu een beslisser B voor A_{LBA} construeren als volgt: bij input
$< M, s >$ doet B het volgende

[4] **Zelf doen**: toon met een voorbeeld aan dat de taal wijzigt!

- berekent $Max = qnb^n$
- simuleert dan M op s voor maximaal Max stappen
- indien M ondertussen accepteerde, accept
- indien M ondertussen verwierp, reject
- indien M nog niet stopte, betekent dat dat M in een lus zit en dus niet zal accepteren: reject

■

Stelling 11.6 $E_{LBA} = \{M | M \textit{ is een LBA die de lege taal bepaalt}\}$ is niet beslisbaar

Bewijs We laten eerst zien dat voor een gegeven Turingmachine M en string s we een LBA kunnen construeren die gegeven een eindige rij configuraties (van M) kan beslissen of die rij een accepterende computation history is voor s. Een rij configuraties kan gemakkelijk op een band geplaatst worden zoals in de figuur hieronder

\$	a	q_4	c	d	\$	a	b	q_7	d	\$

Die stelt een overgang voor waarbij $\delta(q_4, c) = (q_7, b, R)$.

Wat moet een machine doen om na te gaan of een rij configuraties een accepterende computation history is voor s?

- nakijken of twee opeenvolgende configuraties verbonden zijn door de δ
- nakijken of de eerste configuratie q_s bevat op de juiste plaats
- nakijken of de laatste configuratie q_a bevat

Zonder veel in detail te treden moet het duidelijk zijn dat hiervoor slechts een constante hoeveelheid extra bandruimte nodig is en dat die beslissing dus kan genomen worden door een LBA. We maken die LBA zo dat ie bij een accepterende computation history accepteert en anders reject. Nu kunnen we het bewijs zelf beginnen.

Stel dat we een beslisser E hebben voor E_{LBA}. We construeren een beslisser B voor A_{TM} als volgt. Bij input $< M, s >$ doet B:

- construeer de LBA $A_{M,s}$ die van input kan beslissen of een inputstring een accepterende computation history is voor M op input s
- laat E los op $< A_{M,s} >$: als E aanvaardt, reject; anders accept

B beslist A_{TM} want B accepteert $< M, s >$ alss $E < A_{M,s} >$ reject, alss $A_{M,s}$ aanvaardt minstens één string, alss er bestaat een accepterende computation history voor M op s. Het laatste is equivalent met zeggen dat M s accepteert.

Die B kan niet bestaan, dus ook E niet en E_{LBA} is onbeslisbaar. ■

Een klein overzichtje dat overeenkomsten en verschillen tussen PDA, LBA en TM laat zien:

- acceptance en halting
  - PDA en LBA zijn gelijk: acceptance en halting zijn beslisbaar
  - TM verschilt: acceptance en halting zijn niet beslisbaar
- leegheid
  - LBA en TM zijn gelijk: leegheid is niet beslisbaar
  - PDA verschilt: leegheid is beslisbaar

De talen die door een LBA worden bepaald liggen dus tussen de CFL's en de willekeurige talen: zij worden gegenereerd door de context-sensitieve grammatica's.

Als afsluiter (voorlopig): het is niet beslisbaar of een CFG alle strings uit Σ^* genereert, dus $ALL_{CFG} = \{< G > | L_G = \Sigma^*\}$ is niet beslisbaar. Dat is wel opmerkelijk want E_{CFG} is wel beslisbaar.

Stelling 11.7 ALL_{CFG} is niet beslisbaar.

Bewijs Het bewijs wordt niet gezien. ■

Zelf doen: Bewijs dat ALL_{RegExp} beslisbaar is.

12 Wat weetjes over onze talen

Met $Besl$ duiden we de beslisbare talen aan, met $Herk$ de herkenbare. Eerst wat inclusies: zij zijn allemaal strikt.

$$RegLan \subset DCFL \subset CFL \subset Besl \subset Herk \subset \mathcal{P}(\Sigma^*)$$

Nu wat eigenschappen i.v.m. operaties op talen:

$RegLan$ is gesloten onder unie, doorsnede, complement

$DCFL$ is gesloten onder complement, maar CFL niet

CFL is gesloten onder unie, maar $DCFL$ niet

$Besl$ is gesloten onder unie en complement

$Herk$ is gesloten onder unie

Voor de aardigheid eens een niet zo gewone operatie op talen: de omkering. Noteer met $\hat{s}$ de string die je krijgt door de symbolen van s in omgekeerde volgorde te schrijven. Dan is $\widehat{L} = \{\hat{s} | s \in L\}$.

$RegLan$, CFL, $Besl$ en $Herk$ zijn gesloten onder omkering

$DCFL$ is NIET gesloten onder omkering

Voorbeeld: De taal $\{ba^m b^n c^k | m \neq n\} \cup \{ca^m b^n c^k | n \neq k\}$ is deterministisch contextvrij (maak er een CFG voor!) maar zijn omgekeerde is dat niet.

13 Aftelbaar

Dit is een goed moment om op het begrip aftelbaar terug te komen: in het engels spreekt men van countable, maar er bestaat ook de notie van enumerable (of effective enumerable). Dat laatste noemen we *effectief aftelbaar* of *opsombaar*.

Een verzameling V is aftelbaar als er een bijectie bestaat tussen V en (een deel van) $\mathbb{N}$. Die definitie zegt enkel iets over het bestaan van de bijectie, niet over het gemak waarmee die kan berekend worden: wat de definitie betreft hoeft dat zelfs niet.

Nochtans is dat in de context van berekenbaarheid belangrijk, want we hebben al verschillende keren constructies gebruikt zoals *een TM die alle strings van* Σ^* *één voor één genereert*. Om dat mogelijk te maken moet de verzameling effectief aftelbaar zijn. Is het eenvoudig aan te tonen dat er ook aftelbare verzamelingen zijn die niet effectief aftelbaar zijn?

Laat ons eerst het verband zien tussen gegenereerd worden door een enumeratormachine en effectief aftelbaar: de enumeratormachine genereert soms dubbels, maar die kan je vermijden door de machine uit te breiden door in de enumeratortoestand eerst te kijken op de outputband of de nieuwe string wel echt nieuw is. Dus: een enumeratormachine maakt een effectieve aftelling van zijn gegenereerde taal.

Maar elke taal door de enumerator gegenereerd is herkenbaar en we bewezen dat er een taal L bestaat die niet herkenbaar is. Die L kan niet effectief afgeteld worden. Nochtans is die L wel aftelbaar.

Zelf doen: Geef concrete voorbeelden van talen die aftelbaar zijn, maar niet opsombaar.

14 Een dooddoener: de stelling van Rice

We hebben van een aantal concrete talen bewezen dat ze niet beslisbaar zijn, meestal door reductie naar A_{TM}: elke keer vonden we een nieuw truukje om van een veronderstelde beslisser een beslisser te maken voor A_{TM}. De stelling van Rice is dan ook een leukerd: die bewijst dat elke taal (die aan bepaalde voorwaarden voldoet) niet-beslisbaar is. Hier is de context meer precies.

Beschouw de verzameling van Turingmachines (over een vast alfabet als je wil). Een eigenschap P van de Turingmachines verdeelt die in twee verzamelingen: de machines met de eigenschap P ofwel $Pos_P = \{M|P(M) = true\}$, en de machines die de eigenschap niet hebben $Neg_P = \{M|P(M) = false\}$.

Definitie 14.1 Niet-triviale eigenschap
Een eigenschap P van Turingmachines heet *niet-triviaal* indien
$Pos_P \neq \emptyset$ en ook $Neg_P \neq \emptyset$.

Definitie 14.2 Taal-invariante eigenschap
De eigenschap P heet *taal-invariant* indien
$L_{M_1} = L_{M_2} \Longrightarrow P(M_1) = P(M_2)$
of in woorden *alle machines die dezelfde taal bepalen hebben ofwel allemaal P, ofwel heeft geen enkele ervan P.*

Zelf doen:

Geef wat extravagante voorbeelden van taal-invariante niet-triviale eigenschappen van Turingmachines.

Geef wat extravagante voorbeelden van niet-taal-invariante eigenschappen van Turingmachines.

Voor een gegeven niet-triviale taal-invariante eigenschap P, hoeveel machines zijn er die eigenschap P hebben? En niet hebben?

Stelling 14.3 Stelling I van Rice
Voor elke niet-triviale, taal-invariante eigenschap P van Turingmachines geldt dat Pos_P (en ook Neg_P) niet beslisbaar is.

Bewijs Veronderstel dat $M_\emptyset$ (de machine die de lege taal beslist) de eigenschap P niet heeft - indien dat niet zo is, verander dan P in zijn negatie. Vermits P niet-triviaal is bestaat er een taal L_X zodat X een Turingmachine is met de eigenschap P. Stel dat Pos_P (en dus ook Neg_P) beslisbaar is: we zullen een beslisser B voor Pos_P gebruiken om een beslisser A te maken voor A_{TM}. A krijgt als input $< M, s >$ en doet het volgende:

- construeer een hulpmachine $H_{M,s}$ die het volgende doet bij input x:
  - laat M lopen op s
  - indien M s accepteert laat dan X lopen op x en accepteer als X x accepteert
- laat nu B los op $H_{M,s}$
- als B $H_{M,s}$ accepteert, dan accept, anders reject

Kijk eerst na dat je begrijpt dat $H_{M,s}$ ofwel de lege taal accepteert, ofwel L_X. Daarvan maken we gebruik:

A accepteert $< M, s >$ alss B $H_{M,s}$ accepteert, alss $H_{M,s}$ heeft de eigenschap P, alss $H_{M,s}$ accepteert L_X, alss M accepteert s.

Dus, A is een beslisser voor A_{TM}, hetgeen niet kan, dus kan B niet bestaan en Pos_P is niet beslisbaar. ■

Er bestaat een tweede stelling van Rice: elke niet-monotone eigenschap is niet-herkenbaar; de moeite om nader te bekijken!

Zelf doen:

Wat concludeer je nu over de herkenbaarheid van Pos_P en/of Neg_P?

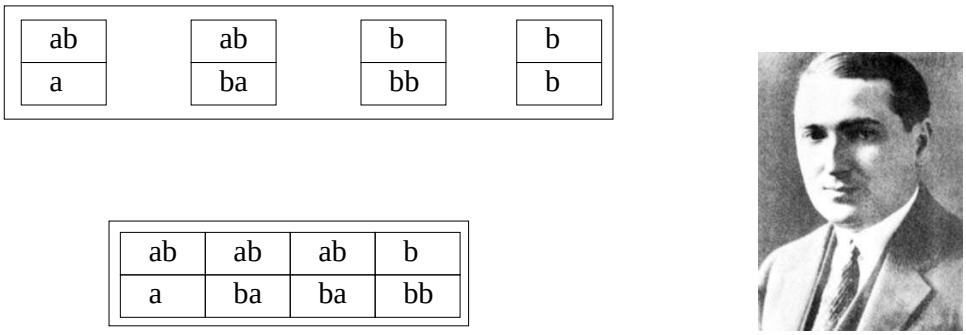
Gebruik Rice voor een nieuw bewijs van vroegere stellingen.

15 Het *Post Correspondence Problem*

Het gaat hier om Emile Post en de *correspondence* heeft niks te maken met het versturen van correspondentie, maar met *overeenkomst*. Emile Post was één van de grondleggers van recursietheorie en ook hij zocht naar universele berekeningsmechanismen. Hij ontwierp het volgende *spel*:

Gegeven een eindig aantal verschillende dominostenen, maar van elk zoveel kopieën als je maar wil. We zullen stenen altijd vertikaal leggen. Op boven- en onderhelft staan strings met tekens uit een gegeven (eindig) alfabet. Kan je een (eindige) rij stenen tegen elkaar leggen zodanig dat de string uit de bovenhelften gelijk is aan de string uit de onderhelften?

Een voorbeeldje:



Figuur 3.4: Vier gegeven stenen en een oplossing

De figuur 3.4 laat zien dat je niet alle stenen hoeft te gebruiken, en dat je een steen meerdere keren mag gebruiken.

Het gaat hier om een beslissingsprobleem: het is genoeg om met zekerheid ja of nee te kunnen antwoorden; het is niet nodig om bij een ja een constructie te maken van een overeenkomst.

Dit simpele *spelletje* blijkt (1) onbeslisbaar en (2) krachtig genoeg om **alle** berekeningen mee te doen die je met een Turingmachine kan doen. Het eerste volgt uit het tweede, van zodra we kunnen laten zien dat *het stoppen van een willekeurige Turingmachine* kan vertaald worden naar *er bestaat een oplossing voor een bepaald PCP*. Die vertaalslag doen we expliciet voor een voorbeeld: de generalisatie kan je vinden o.a. in Sipser of Minsky.

15.1 Van Turingmachine naar het Postspelletje

In dit concreet voorbeeld gebruiken we X voor de starttoestand, A voor accepterende toestand, Z voor de reject toestand. Het alfabet is $\{a, b\}$, en de transitietabel is

1	X	a	B	a	R
2	X	b	Z	_	_
3	X	#	A	#	S
4	B	a	Z	_	_
5	B	b	X	b	R
6	B	#	Z	_	_

De nummering dient om er later gemakkelijk naar te kunnen verwijzen. Kijk na dat deze tabel een Turingmachine specifieert die de taal $(ab)^*$ accepteert. Stel dat de input waarop we de machine laten lopen ab is. We voeren nog een nieuw symbool in: \$. We geven nu de stenen die we nodig hebben om met een Postspel die Turingmachine na te bootsen:

- voor elk symbool x maak een steen met van boven en van onder x, dus de 4 stenen: $\begin{array}{|c|c|c|c|}\hline a & b & \$ & \# \\ \hline a & b & \$ & \# \\ \hline \end{array}$ en stenen met Ax of xA van boven en A van onder, dus 8 stenen: $\begin{array}{|c|c|c|c|}\hline aA & bA & \$A & \#A \\ \hline A & A & A & A \\ \hline \end{array}$ $\begin{array}{|c|c|c|c|}\hline Aa & Ab & A\$ & A\# \\ \hline A & A & A & A \\ \hline \end{array}$
- voor regel 1 in de transitietabel maken we een steen $\begin{array}{|c|}\hline Xa \\ \hline aB \\ \hline \end{array}$
- voor regel 5 in de transitietabel maken we een steen $\begin{array}{|c|}\hline Bb \\ \hline bX \\ \hline \end{array}$
- voor regel 3 maken we $\begin{array}{|c|}\hline X\# \\ \hline A\# \\ \hline \end{array}$
- de volgende stenen hangen niet af van de gegeven Turingmachine: de afsluiter $\begin{array}{|c|}\hline A\$\$ \\ \hline \$ \\ \hline \end{array}$ en de hekjes generatoren links en rechts: $\begin{array}{|c|c|}\hline \$ & \$ \\ \hline \$\# & \#\$ \\ \hline \end{array}$
- en tenslotte gieten we de input ab in de steen $\begin{array}{|c|}\hline \$ \\ \hline \$Xab\$ \\ \hline \end{array}$: een beginconfiguratie.

We vragen nu: construeer een correspondentie met deze stenen beginnend met de inputsteen.[5] Hier is de oplossing:

$	Xa	b	$	a	Bb	#	$	a	b	X#	$	a	bA	#	$	aA	#	$	A#	$	A$$
Xab	aB	b	#$	a	bX	#	$	a	b	A#	$	a	A	#	$	A	#	$	A	$	$

In de onderste lijn zie je tussen twee $-tekens telkens een configuratie: twee opeenvolgende configuraties zijn verbonden door de transitiefunctie, tot op het ogenblik dat de accepterende toestand verschijnt. Daarna wordt de configuratie leeg gemaakt tot daarin alleen nog de accepterende toestand staat, en dan volgt de afsluiter.

Zelf doen:

Overtuig je er nu van dat als je begint te bouwen met een input die geen string is in de taal $(ab)^*$, je geen correspondentie kan vinden.

Zoek het algemene verband tussen δ en de stenen: hierboven hadden we geen bewegingen van de leeskop naar links!

Formuleer nu nauwkeuriger het verband tussen PCP en H_{TM}.

[5] Het algemene PCP laat toe om met een willekeurige steen te beginnen, maar je kan het ene in het andere transformeren.

Een extraatje: de Post tag machine E. Post staat ook bekend om zijn *tag systems* of *Post tag machines*. Laten we een klein voorbeeld geven van een 2-tag systeem:

- het alfabet is $\{a, b, c\}$ en er is een haltsymbool H
- de regels zijn
  - a $\rightarrow$ aa
  - b $\rightarrow$ accH
  - c $\rightarrow$ a
- het initieel woord is aab
- en hier is een herschrijfrij

  aab $\rightarrow$ baa $\rightarrow$ aaccH $\rightarrow$ ccHaa $\rightarrow$ Haaa

De algemene regel voor herschrijven is: de eerste letter van het woord bepaalt welke regel zal gebruikt worden (er kunnen meerdere regels zijn voor dat symbool indien het een niet-deterministisch tag systeem is). Schrijf de rechterkant van de regel vanachter bij de string en veeg van voor 2 symbolen uit - dezelfde 2 als in 2-tag systeem.

Als er een H links staat is de berekening gedaan en kan je de string beschouwen als het resultaat van de berekening bij input de initiële string.

Ook dit herschrijfsysteem is Turingcompleet.

2-tag systems zijn belangrijk o.a. omdat ze met kleine UTM's kunnen gesimuleerd worden.

16 Veel-één reductie

Je hebt in je inleiding tot complexiteitstheorie al kennis gemaakt met het begrip *is polynoom reduceerbaar naar*. Die relatie gaf inzichten in de structuur van P, en liet de definitie van NP-compleet toe. Als een taal L_1 polynoom naar L_2 kan gereduceerd worden, dan weten we ook dat L_2 in zekere zin *moeilijker is*.

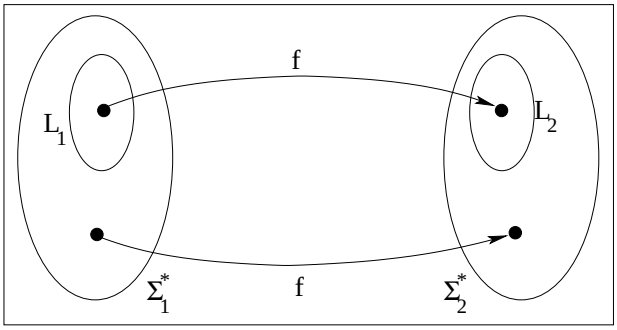
In de context van berekenbaarheid bestaat een gelijkaardige reductie van talen. Vermits complexiteit nu niet belangrijk is, laten we zeker de eis van polynomialiteit vallen: we vervangen ze door Turing-berekenbaarheid.

Definitie 16.1 Turing-berekenbare functie
Een functie f heet Turing-berekenbaar indien er een Turingmachine bestaat die bij input s uiteindelijk stopt met $f(s)$ op de band.

We hebben niet geëist dat de machine in de accepterend eindtoestand stopt, maar dat zou niks essentieels veranderen. Dikwijls kort men Turing-berekenbaar af tot berekenbaar.

Definitie 16.2 Reductie van talen
We zeggen dat taal L_1 (over Σ_1) naar taal L_2 (over Σ_2) kan gereduceerd worden indien er een afbeelding f met signatuur $\Sigma_1^* \to \Sigma_2^*$ bestaat zodanig dat $f(L_1) \subseteq L_2$ en $f(\overline{L_1}) \subseteq \overline{L_2}$, en zodanig dat f Turing-berekenbaar is.
We noteren dat door $L_1 \leq_m L_2$.

In het engels heet zulk een reductie een *many-one reduction* en reduceerbaar wordt *(mapping) reducible* genoemd. Figuur 3.5 toont de eerste voorwaarde in de definitie.



Figuur 3.5: Schematische voorstelling van een reductie

Een reductie f geeft een manier om vragen over L_1 om te zetten naar vragen over L_2, in het bijzonder vragen van de vorm $s \in L_1$? Inderdaad, om te testen of $s \in L_1$, kunnen we volstaan met testen of $f(s) \in L_2$. De volgende stellingen en gevolg geven wat verbanden tussen twee zulke talen: bewijs ze zelf.

Stelling 16.3 Als $L_1 \leq_m L_2$ en L_2 is beslisbaar, dan is L_1 beslisbaar.

Stelling 16.4 Als $L_1 \leq_m L_2$ en L_2 is herkenbaar, dan is L_1 herkenbaar.

Gevolg 16.5 Als $L_1 \leq_m L_2$ en L_1 is niet-herkenbaar, dan is L_2 niet-herkenbaar. Als $L_1 \leq_m L_2$ en L_1 is niet-beslisbaar, dan is L_2 niet-beslisbaar.

We hebben vroeger al één keer een mapping reductie gebruikt, zij het informeel: in de stelling op pagina 105 toen we bewezen dat EQ_{TM} niet beslisbaar is. We doen het hier meer formeel

Stelling 16.6 EQ_{TM} is niet beslisbaar.

Bewijs We weten al dat E_{TM} niet beslisbaar is, en we
reduceren nu E_{TM} naar EQ_{TM} met de functie f die bij input $< M >$ als resul-

taat geeft: $< M, M_\phi >$ waarin M_ϕ een Turingmachine is die de lege taal bepaalt. Het is duidelijk dat f Turing-berekenbaar is.

Bijgevolg $E_{TM} \leq_m EQ_{TM}$ en we kunnen vorig gevolg gebruiken. ■

Stelling 16.7 Als $A \leq_m B$ dan ook $\overline{A} \leq_m \overline{B}$.

Bewijs Zelf doen! ■

We hebben vroeger ook al andere *reducties* gemaakt van één taal naar een andere: het bewijs van de stelling op pagina 104 gebruikt de onbeslisbaarheid van A_{TM} om die van E_{TM} aan te tonen.[6] Die stelling maakte wel een mapping reductie van $\overline{A_{TM}}$ naar E_{TM}, als volgt: met $< M, s >$ laten we overeenkomen $< M_s >$ (zie het bewijs van die stelling)...

Stelling 16.8 EQ_{TM} is niet herkenbaar en ook niet co-herkenbaar.

Bewijs We zullen twee mapping reducties construeren, de eerste zal aantonen dat $A_{TM} \leq_m \overline{EQ_{TM}}$ en de andere toont aan dat $A_{TM} \leq_m EQ_{TM}$. Vermits $\overline{A_{TM}}$ niet herkenbaar is, bewijst dat de stelling.

1. f transformeert $< M, s >$ in $< M_s, M_\phi >$; daarin is M_s een machine die elke string accepteert indien M s accepteert; het is duidelijk dat f berekenbaar is; we moeten nog de andere voorwaarden aantonen:
   - indien M s accepteert, dan zijn M_s en M_ϕ verschillend
   - indien M s niet accepteert, dan zijn M_s en M_ϕ gelijk
2. f transformeert $< M, s >$ in $< M_s, M_{\Sigma^*} >$; M_{Σ^*} is de machine die alles accepteert; M_s is zoals hiervoor; de voorwaarden op f zijn gemakkelijk na te gaan

■

In volgend hoofdstuk bekijken we een tweede manier om talen te reduceren.

[6] Maar een mapping reductie van A_{TM} naar E_{TM} bestaat niet - probeer dat te bewijzen of in te zien.

17 Orakelmachines en een hiërarchie van beslisbaarheid

Stel dat we een manier hadden om A_{TM} te beslissen, kunnen we dan alles beslissen? Laten we die vraag eerst wat concreter maken:

- een manier om A_{TM} te beslissen kan niet een Turingmachine zijn; we moeten het dus een nieuwe naam geven: een *orakel*; we zien later meer concreet hoe een orakel kan gerealiseerd worden ...
- het orakel moet kunnen geraadpleegd worden door een Turingmachine; meer concreet, het orakel voor A_{TM} moet door een Turingmachine kunnen opgeroepen worden als een subroutine, met een string als input voor het orakel; het orakel heeft slechts een eindig aantal stappen nodig om de beslissing aan de Turingmachine mee te delen;
- op die manier hebben we een orakelmachine $O^{A_{TM}}$ gebouwd: een Turingmachine die vragen kan stellen aan het orakel voor A_{TM}

Het is duidelijk dat we een $O^{A_{TM}}$ kunnen maken die A_{TM} beslist: geef de input $< M, s >$ aan het orakel, en geef als antwoord wat het orakel zegt. Het is dus direct duidelijk dat de verzameling van orakelmachines met orakel A_{TM} strikt sterker is dan de verzameling Turingmachines. Hier nog een voorbeeld:

Stelling 17.1 Er bestaat een $O^{A_{TM}}$ die E_{TM} beslist.

Bewijs We construeren $O^{A_{TM}}$ als volgt: bij input $< M >$ doet $O^{A_{TM}}$

- construeer een Turingmachine P die bij input w doet:
  - laat M lopen op alle strings van Σ^* (*)
  - als M een string accepteert, accept
- vraag aan het orakel voor A_{TM} of $< P, x > \in A_{TM}$
- als het orakel **ja** antwoordt, reject; anders accept

Als $L_M \neq \emptyset$, dan accepteert P elke input, en zeker input x; dus antwoordt het orakel **ja** en dus moet $O^{A_{TM}}$ verwerpen. Omgekeerd: als $L_M = \emptyset$, dan accepteert $O^{A_{TM}}$. We besluiten dat $O^{A_{TM}}$ de taal E_{TM} beslist. ■

Verdergaand op vorige stelling: we zeggen dat E_{TM} beslisbaar is relatief t.o.v. A_{TM}. Dit brengt ons bij de definitie:

Definitie 17.2 Turingreduceerbaar
Een taal A is Turingreduceerbaar naar taal B, indien A beslisbaar is relatief t.o.v. B, t.t.z. er bestaat een orakelmachine O^B die A beslist. We schrijven $A \leq_T B$.

Die definitie sluit aan bij onze intuïtie over wat reduceerbaarheid zou moeten betekenen:

Stelling 17.3 Indien $A \leq_T B$ en B is beslisbaar, dan is A beslisbaar.

Het bewijs hiervan - en ook van de volgende stelling - doe je zelf.

Stelling 17.4 Indien $A \leq_m B$ dan is ook $A \leq_T B$.
M.a.w. $\leq_m$ is fijner dan $\leq_T$.

Hier een beschrijving van een orakel voor L: elke string heeft een volgnummer in de lexicografische orde met kortere strings eerst. De taal kan voorgesteld worden als een bitmap, waarbij op de i-de plaats een 1 staat als de i-de string in L zit, en anders een 0. Die bitmap kunnen we op een band zetten. Als het orakel een vraag krijgt over een string s, dan berekent het orakel het volgnummer i van s, en kijkt wat er op de i-de plaats van de bitmapband staat.

Kan de klasse van orakelmachines met een orakel voor A_{TM} elke taal beslissen? Nee. Het aantal talen is niet-aftelbaar; het aantal orakelmachines met het orakel voor A_{TM} is aftelbaar. Dus bestaat er een taal X die niet beslisbaar is m.b.v. het orakel voor A_{TM}. Je kan nu die redenering herhalen met een orakel voor X en je merkt dat er een hele hiërarchie bestaat van steeds moeilijker te beslissen talen. Ook in de context van complexiteitstheorie worden orakelmachines gedefinieerd. Dat geeft ook aanleiding tot een hiërarchie. Die kan nog altijd instorten als blijkt dat $NP = P$. Onze berekenbaarheidshiërarchie blijft echter zeker overeind.

Zelf doen: In de stelling op pagina 120 staat een (*): hoe kan M lopen op alle strings? Dat zijn er oneindig veel!

18 Turing-berekenbare functies en recursieve functies

We hebben in een vorig hoofdstuk een definitie gegeven van de Turing-berekenbare functies. Die was nogal abstract en vooral existentieel. Een andere manier om greep te krijgen op welke functies door een Turingmachine berekenbaar zijn, is door *bottom-up* te werken: we beginnen met eenvoudige functies, stellen die samen met eenvoudige operatoren en zien hoe ver we geraken. Deze weg werd bewandeld door Kurt Goedel en Jacques Herbrand.

18.1 Primitief recursieve functies

Basisfuncties

- de nulfunctie: $nul : \mathbb{N} \to \mathbb{N}$
$$nul(x) = 0$$
- de successorfunctie: $succ : \mathbb{N} \to \mathbb{N}$
$$succ(x) = x + 1$$
- projecties: ${p_i}^n : \mathbb{N}^n \to \mathbb{N}$
$${p_i}^n(x_1, x_2, ..., x_n) = x_i$$

Compositie

Gegeven:

- $g_1, g_2, ..., g_m$ functies $\mathbb{N}^k \to \mathbb{N}$
- f functie $\mathbb{N}^m \to \mathbb{N}$

maak door compositie functie $h : \mathbb{N}^k \to \mathbb{N}$:
$$h(\overline{x}) = f(g_1(\overline{x}), g_2(\overline{x}), ..., g_m(\overline{x}))^7$$
Notatie: $h = Cn[f, g_1, \ldots, g_m]$

Primitieve recursie

Gegeven:

- $f : \mathbb{N}^k \to \mathbb{N}$
- $g : \mathbb{N}^{k+2} \to \mathbb{N}$

[7] $(\overline{x} = x_1, x_2, ..., x_k)$

maak door primitieve recursie $h : \mathbb{N}^{k+1} \to \mathbb{N}$:

$$h(\overline{x}, 0) = f(\overline{x})$$
$$h(\overline{x}, y+1) = g(\overline{x}, y, h(\overline{x}, y))$$

Notatie: $h = Pr[f, g]$

Bovenstaande geldt voor $k > 0$. Voor $k = 0$ is het: $h : \mathbb{N} \to \mathbb{N}$:

$$h(0) = c \text{ waarbij } c \text{ een getal is}$$
$$h(y+1) = g(y, h(y))$$

Definitie 18.1 Primitief recursieve functie
Alle functies die je kan maken door te vertrekken van de basisfuncties en door gebruikmaking van compositie en primitieve recursie, noemen we **primitief recursief**.

Primitief recursieve functies zijn overal gedefinieerd. Ze kunnen berekend worden met for-programma's.

Voorbeelden

- $Cn[succ, nul]$ is de constante functie 1
- $Cn[succ, Cn[succ, Cn[succ, nul]]]$ is de constante functie 3
- $Pr[{p_1}^1, Cn[succ, {p_3}^3]]$ = som van 2 inputs ($\mathbb{N}^2 \to \mathbb{N}$)
  schrijf uit en zie analogie met:
  som(x,0) = x
  som(x,y+1) = som(x,y) + 1
- $Pr[nul, Cn[som, {p_1}^3, {p_3}^3]]$ = product van 2 inputs ($\mathbb{N}^2 \to \mathbb{N}$)
- faculteitsfunctie, predecessor ...

18.2 Recursieve functies

Een niet primitief-recursieve functie: de Ackermann functie

Goedel's originele constructie van *alle berekenbare functies* bevatte oorspronkelijk alleen maar de primitief-recursieve functies. Wilhelm Ackermann, student van David Hilbert, publiceerde de nu bekende Ackermann functie in 1928: die functie

is duidelijk *berekenbaar* volgens een intuïtief begrip van berekenbaarheid (en ook op een Turingmachine, maar bestond die al in 1928?), maar is niet primitief-recursief. Voor de historische noot: een andere student van Hilbert, Gabriel Sudan, was eigenlijk de eerste die een berekenbare, niet primitief-recursieve functie ontdekte. Hier is de definitie van de Ackermann functie in twee veranderlijken.

$$\text{Ack}(0,\ y\ ) = y + 1$$

$$\text{Ack}(x + 1,\ 0) = \text{Ack}(x,\ 1)$$

$$\text{Ack}(x + 1,\ y + 1) = \text{Ack}(x,\ \text{Ack}(x + 1,\ y\ ))$$

Deze functie is overal gedefinieerd (probeer dat in te zien!) en stijgt sneller dan elke primitief recursieve functie: daarom is ze niet primitief recursief.

Dikwijls wordt met Ackermann's functie een functie in één argument bedoeld, gedefinieerd als volgt: `Ack(n) = Ack(n,n)`. Deze functie stijgt zeer snel: ook sneller dan elke primitief-recursieve functe in één veranderlijke. Zijn inverse is o.a. van belang in complexiteitsanalyse: zoek het Union-Find algoritme.

Onbegrensde minimizatie

Goedel moest dus zijn klasse van functies uitbreiden om ook de Ackermann functie toe te laten. Hij deed dat door *onbegrensde minimizatie* te introduceren.
Gegeven

- $f : \mathbb{N}^{k+1} \to \mathbb{N}$

Construeer door onbegrensde minimisatie:
$g : \mathbb{N}^k \to \mathbb{N}$ als volgt:

$g(\overline{x}) = y$ als

$f(\overline{x}, y) = 0$ en
$f(\overline{x}, z)$ is gedefinieerd voor alle $z < y$ en $f(\overline{x}, z) \neq 0$

en anders is $g(\overline{x})$ niet gedefinieerd

Notatie: $g = Mn[f]$
g geeft minimale nulpunten van f

Code om Mn[f](x) te berekenen

```
y = 0;
while (f(x,y) != 0)
   y++;
return y;
```

Je kan op twee manieren in een lus terecht komen ...

Definitie 18.2 Recursieve functies
Recursieve functies verkrijg je uit de basisfuncties en toepassen van Pr, Cn en Mn. Die functies worden ook *μ-recursive* genoemd.

Deze functies kunnen met while-programma's berekend worden. Meer bepaald kan elke recursieve functie berekend worden met een Turingmachine en omgekeerd. Dus: de Turing-berekenbare functies zijn exact de recursieve functies. Het is duidelijk uit de definitie van onbegrensde minimisatie dat recursieve functies *partieel* kunnen zijn. Dat is consistent met Turing-berekenbaar, vermits ook een TM dikwijls slechts een partiële functie definieert. De Ackermann functie laat zien dat een echte recursieve functie ook totaal kan zijn.

Zelf doen:

Als het domein van een partiële recursieve functie beslisbaar is, dan kan de functie uitgebreid worden tot een totale functie, die ook recursief is: probeer dat in te zien, of nog beter, te bewijzen.

Bestaat er een partiële recursieve functie waarvan het domein niet beslisbaar is?

Is het domein van een (partiële) recursieve functie herkenbaar?

Is het bereik van een (partiële) recursieve functie herkenbaar?

19 De bezige bever en snel stijgende functies

De Ackermann functie liet al zien dat snel stijgen een aanduiding kan zijn dat er extra machinerie nodig is om de functie nog te kunnen berekenen. Voor de Ackermann functie lukte dat nog met machinerie die door een Turingmachine kan geïmplementeerd worden: onbegrensde minimizatie. De essentie daarvan is *zoek het* **kleinste** *getal dat aan een voorwaarde V voldoet*. Dat kan je implementeren door:

$i = 0;$
$while\ not(V(i))\ i++;$

en je weet al dat dat niet eindigt als geen enkele i voldoet aan V.

Stel dat we nu als nieuwigheid invoeren *onbegrensde maximalisatie*, t.t.z. *zoek het* **grootste** *getal dat voldoet aan voorwaarde V*. Hoe zou dat kunnen geïmplementeerd worden? Hier een eerste poging:

$i = \infty;$
$while\ not(V(i))\ i--;$

Dat werkt niet - argumenteer zelf.

Hier een tweede poging:

$i = 0;$
$while\ i < \infty$
    $if\ V(i)\ max = i;$
    $i++;$

Ook die heeft problemen ...

Dit laat zien dat het invoeren van een nieuwe constructie om een grotere klasse functies te kunnen berekenen, niet zonder gevaar is.

19.1 De bezige bever

Beschouw alle Turingmachines met alfabet $\{0, 1\}$ en n toestanden. Zo zijn er maar een eindig aantal. We kunnen die Turingmachines laten lopen met als input een lege band, en als de machine stopt (in q_a of q_r maakt niet uit) tellen hoeveel keer er een 1 op de band staat. Een bezige bever is een kampioen in zijn klasse, t.t.z. voor een gegeven n bestaat geen andere machine in dezelfde klasse

die **meer** enen uitschrijft en stopt. We noteren het aantal enen uitgeschreven door een bezige bever van klasse n met $\Sigma(n)$. Sorry voor de verwarring met onze gewone notatie voor een alfabet, maar Tibor Radó die de bezige bever uitvond noteerde het ook zo.

Σ definieert een totale functie met signatuur $\mathbb{N} \to \mathbb{N}$. De vraag *is* Σ *Turing-berekenbaar* is dus aan de orde.

Voor kleine n lijkt het te doen om een bezige bever te construeren - door exhaustief alle TM's te maken met n toestanden, die te laten lopen op een lege input en achteraf het aantal enen te tellen.

Maar er is een probleem: door het Halting-probleem weten we dat we niet altijd kunnen weten of een machine stopt op de lege input. Dus moeten we van de machines die al heel lang lopen, telkens een apart bewijs geven dat ze eigenlijk in een lus zitten, ofwel kunnen we alleen maar ondergrenzen vinden voor $\Sigma(n)$. En vergis je niet: men denkt dat er kleine universele Turingmachines bestaan, dus al bij heel kleine n zijn er machines waarvoor het niet-triviaal is sommige eigenschappen te bewijzen.

De waarden van $\Sigma(n)$ zijn exact bekend voor $n < 5$. Voor $n = 5$ is het huidige record 4098, maar van een aantal machines (die nog niet gedaan hebben :-) is niet geweten of ze in een lus zitten: die machines zouden dus nog het record kunnen verbeteren. Een gelijkaardige situatie bij $n = 6$ waar het record op 10^{1730} staat.

Σ is een functie die verkregen wordt door *begrensde* maximalisatie, maar wel over een niet-berekenbare verzameling: die verzameling bevat juist de stoppende Turingmachines (met n toestanden). Dat vormt de inherente reden waarom Σ niet berekenbaar is, al bestaat de functie overal en is ze abstract goed gedefinieerd.

Veel open problemen uit de wiskunde zouden *eenvoudig* opgelost kunnen worden als we Σ exact kenden voor sommige n.

Zelf doen: Welke uitspraken zijn juist? Graag argumentatie bij je antwoorden!

$\Sigma(n)$ kan bepaald worden voor elke n, maar is ∞ voor n groot genoeg.

Het is mogelijk dat voor n groot genoeg $Ack(n) > \Sigma(n)$.

Elke functie die trager stijgt dan een gegeven primitief-recursieve functie is zelf primitief-recursief.

Hoofdstuk 4

Herschrijfsystemen

1 Inleiding

1.1 Een klein herschrijfsysteem

Informeel kunnen we een herschrijfsysteem definiëren als een verzameling (syntactische) termen en een verzameling (herschrijf)regels die toelaten om termen te herschrijven. Een eenvoudig voorbeeld van een herschrijfsysteem is $(Plus, R)$:

- De verzameling $Plus$ kan men recursief als volgt bepalen:
  1. alle natuurlijke getallen behoren tot $Plus$
  2. als x, y behoren tot $Plus$, dan behoort ook de formele uitdrukking $(x + y)$ tot Plus.
- $R = \{$gebruik commutativiteit, associativiteit, optelling, substitutie$\}$

Met behulp van R kan je dan de term $(3 + (1 + 1))$ herschrijven tot $(3 + 2)$ of tot 5 of $(2 + 3)$ of $((1 + 3) + 1)$ ... wat we dan noteren als

- $(3 + (1 + 1)) \to (3 + 2)$ (optelling)
- $(3 + (1 + 1)) \to ((1 + 1) + 3)$ (commutativiteit)
- $(3 + (1 + 1))$
  $\to ((3 + 1) + 1))$ (associativiteit)
  $\to (4 + 1)$ (optelling)
  $\to 5$ (optelling)

Bij het laatste voorbeeld hebben we meerdere herschrijfregels nodig om tot het eindresultaat te komen.

Er zijn nog andere manieren om $(3 + (1 + 1))$ te herschrijven tot 5, bijvoorbeeld:
$(3 + (1 + 1)) \to (3 + 2) \to 5$

Dus, hetzelfde *resultaat* kan op meer dan één manier bekomen worden.

Sommige termen zien er eenvoudiger of kleiner uit dan andere; bijvoorbeeld 14 is eenvoudiger dan
$((3 + 2) + 7) + 2)$.
en is in zekere zin minimaal.

Direct rijzen er vragen zoals

- bestaat er een herschrijfrij van de ene gegeven term tot de andere?
- bestaat er een kortste herschrijfrij?
- welke is de complexiteit van die herschrijfrij?
- welke is de complexiteit van het vinden van die herschrijfrij?
- heeft elke term een *minimale voorstelling*?
- heeft elke term een *unieke* minimale voorstelling?

1.2 $(Plus, R)$ meer formeel.

We formaliseren eerst $(Plus, R)$ door de herschrijfregels R uit te drukken op een meer formele manier:

$\forall A, X, Y, Z \in Plus$:

- **commutativiteit**:

  (C) $(X + Y) \rightarrow (Y + X)$

- **associativiteit**:

  (A) $((X + Y) + Z) \rightarrow (X + (Y + Z))$

- **optelling**:

  (O) $(0{+}1) \leftrightarrow 1$
  $(0{+}2) \leftrightarrow 2$
  $\ldots$
  $(1{+}1) \leftrightarrow 2$
  $(1{+}2) \leftrightarrow 3$
  $\ldots$

- **substitutie**:

(S) indien X $\to$ Y dan ook $(A + X) \to (A + Y)$ en $(X + A) \to (Y + A)$

Herschrijfregels kunnen enkel gebruikt worden in de richting van de pijl. Voor $(Plus, R)$ is dat geen echte beperking: (C) is al symmetrisch; van (A) kunnen we bewijzen (dankzij commutativiteit) dat (A) ook omgekeerd mag gebruikt worden en de regels voor de optelling zijn al gedefinieerd met een dubbele pijl.

De S-regel is een concretisering van een algemene herschrijfregel die zegt dat deelexpressies mogen vervangen worden door er een andere herschrijfregel op toe te passen. In het algemeen mag de S-regel niet omgekeerd gebruikt worden.

De optelling is gespecifieerd door een oneindig aantal herschrijfregels.

We noemen een term **minimaal** als de term geen + teken bevat.

We noemen twee termen A en B **equivalent** $(A \sim B)$ als er een term X bestaat en herschrijfrijen zodanig dat

A $\to \ldots \to$ X

en ook

B $\to \ldots \to$ X

Dat $\sim$ een equivalentierelatie is, steunt op het feit dat elke term herschreven kan worden tot een uniek natuurlijk getal. Er volgt dan ook dat $\sim$ de kleinste equivalentierelatie is die $\to$ bevat.

Binnen het herschrijfsysteem $(Plus, R)$ is het nu niet moeilijk om aan te tonen dat

- de equivalentieklasse van elke term exact één minimale term bevat
- elke term herschreven kan worden tot de minimale term in zijn equivalentieklasse met een eindige herschrijfrij

Dat is prachtig, maar *hoe vinden we die unieke minimale term in de equivalentieklasse van een term?*

We hebben een zekere methode nodig om de herschrijfregels toe te passen: zulk een methode wordt een *strategie* genoemd. Een eenvoudige strategie zou er in bestaan om gewoon de (lexicaal) eerste toepasbare herschrijfregel toe te passen. Als een term een + bevat kan men altijd (C) toepassen: het probleem is dat op die manier telkens een oneindige herschrijfrij gemaakt wordt die niet leidt tot de minimale term:

(5+3) $\to$ (3+5) $\to$ (5+3) $\to \ldots$

We hebben dus een betere strategie nodig: bijvoorbeeld de strategie die enkel regel O gebruikt (in combinatie met S) en slechts in één richting.

Een andere mogelijkheid is om een meer beperkt herschrijfsysteem te gebruiken, waarin enkel S en O voorkomen met voor O enkel gebruik van de regels van links naar rechts, t.t.z.

$$(5+3) \rightarrow 8$$

mag, maar

$$8 \rightarrow (5+3)$$

mag niet. Als strategie - op de beperkte regels - nemen we nu: *pas toe wat je kan*. Deze strategie vindt vertrekkend van gelijk welke term een minimale term. In het bijzonder:

- elke herschrijfrij (volgens de strategie) stopt (omdat uiteindelijk geen enkele regel meer toepasbaar is)
- elke herschrijfrij stopt met een minimale term

We kunnen nu gefundeerd zeggen dat de *betekenis* van een term gelijk is aan de unieke minimale term die ermee equivalent is.

Merk op dat de strategie *pas toe wat je kan* **niet-deterministisch** is: in de term $((1+2)+(3+4))$ kan je op twee plaatsen de S-regel toepassen; het niet-determinisme is van die aard dat het niet uitmaakt welke keuze je maakt: de uiteindelijke minimale term is altijd 10[1]. Je mag zelfs de twee toepassingen van de S-regel gelijktijdig doen.

We willen er nog eens de aandacht op trekken dat door een aantal herschrijfregels of hun richting uit te sluiten, we gemakkelijk een strategie konden definiëren met bovenstaande eigenschappen; nochtans is het uitsluiten van herschrijfregels (of van hun gebruiksrichting) niet a priori verdedigbaar: commutativiteit geldt voor de optelling en $3 = (1+2)$ zodat het wel wat argumentatie vraagt om die weg te laten. Bovendien is het gebruik van commutativiteit *nodig* om voor een veranderlijke x de term $1 + x - 1$ te kunnen herleiden tot x.

Waarom is dit interessant?

Dat alles lijkt veel werk voor een triviaal resultaat, maar bekijk het van het standpunt van iemand die een pakket ontwerpt voor het manipuleren van algebraïsche expressies en je merkt dat zelfs dit eenvoudige voorbeeld meer dan een denkoefening is: in zulk een pakket zit impliciet of expliciet een beschrijving van de termen waarmee het pakket rekent, de rekenregels, de equivalentie tussen

[1] die eigenschap heeft te maken met het begrip *confluentie*

termen .... Bedenk daarbij ook dat in het algemeen arithmetische expressies meerdere operatoren hebben (vermenigvuldiging, deling ...) en dat sommige herschrijfregels meerdere operatoren toelaten (distributiviteit van + t.o.v. $*$ ...). De ontwerper moet ook het begrip *minimaliteit* in het pakket implementeren en een strategie kiezen die ervoor zorgt dat minimale termen gevonden worden. Het is daarbij zelfs niet altijd duidelijk welke term minimaal is: is bijvoorbeeld $2x/(x^2-1)$ *kleiner* dan $1/(x+1)+1/(x-1)$?

Een ander aspect van herschrijfsystemen is hun berekeningskracht: wat kan met het herschrijfsysteem berekend worden? Op het eerste zicht is dat een rare vraag, maar in de volgende sectie zullen we *Plus* gebruiken om dingen te berekenen.

1.3 Van *Plus* tot functies

Allereerst breiden we de verzameling termen uit tot

$Plus = \{$ arithmetische expressies met +, natuurlijke getallen en variabelen $\}$

We noteren variabelen altijd met kleine letters. Dus

- x+1
- (3+x+y)+4

behoren tot *Plus*.

Vervolgens beschouwen we een element E van *Plus* met variabelen $x_1 \ldots x_n$ als een functie van $\mathbb{N}^n \to \mathbb{N}$ als volgt:

$$E(i_1, \ldots, i_n) = \text{de minimale term equivalent met } E_{\{x_1 \backslash i_1, \ldots, x_n \backslash i_n\}}$$

Hierin betekent $E_{\{x_1 \backslash i_1, \ldots, x_n \backslash i_n\}}$ de term E waarin elk voorkomen van x_1 vervangen is door i_1, x_2 door i_2 enzovoort.
We weten al dat zulk een minimaal element uniek is en dus is de functie goed gedefinieerd. Neem als voorbeeld $E = (x+7)$, dan is $E(9)$ gelijk aan 16.

We hebben hier dus een mechanisme om functies te definiëren over $\mathbb{N}^n$ en hierin ligt het verband met programmeren: een heleboel programmeerwerk bestaat hoofdzakelijk in het definiëren van functies.

Dat leidt tot de vraag: kunnen we *Plus* echt gebruiken om in te programmeren? Is *Plus* krachtig genoeg? Daarvoor moeten we o.a. weten wat de klasse van functies in één veranderlijke is die we kunnen definiëren m.b.v. *Plus*? Je kan

nagaan dat die klasse heel klein is: alleen maar de lineaire functies f van de vorm:

$f(x) = n * x + m$ waarbij n en m natuurlijke getallen zijn

Bijvoorbeeld de functie die het verband tussen graden Celsius en graden Fahrenheit aangeeft, kan niet uitgedrukt worden in $Plus$, laat staan de functies $faculteit$ of $Fibonacci$. $Plus$ is wel elegant omdat het wat mooie eigenschappen heeft (zoals een volledige strategie, t.t.z. een strategie die altijd een minimale term vindt) maar $Plus$ is niet zo krachtig als een Turingmachine en zeker te beperkt voor het meeste programmeerwerk.

Afgezien van bovenstaande vragen, zijn er nog andere aspecten aan herschrijfsystemen die het bekijken waard zijn

- kunnen functies samengesteld worden? (binnen het formalisme uiteraard)
- kan een functie gedefinieerd worden in functie van een andere?
- kan een functie als resultaat een functie hebben?
- kan een functie als argument een functie hebben?
- ...

$Plus$ heeft niet de kracht en de goede notatie om dit allemaal zomaar toe te laten. Deze aspecten komen aan bod in sectie 2.

1.4 Java en herschrijfsystemen

We bekijken nu een herschrijfsysteem dat we $Java$ zullen noemen: elke term in $Java$ is een drietal

$(programma,\ instructieteller,\ varassoc)$

waarbij *programma* de rij instructies is uit de *main* methode van een Java programma zonder andere methoden en *instructieteller* een teller die een instructie uit het programma aanduidt; *varassoc* is een verzameling van associaties tussen een programmaveranderlijke en een waarde. We zullen voor de eenvoud ook veronderstellen dat elk programma eindigt met een *exit(0)*-instructie.

Er zijn in feite zoveel herschrijfregels als er Java-instructies zijn, maar we kunnen die samenvatten in één herschrijfregel die informeel uitgedrukt wordt als:

$(prog, P, varassoc) \rightarrow (prog, P', varassoc')$

waarbij *varassoc'* verkregen wordt uit *varassoc* door de instructie waar P naar wijst uit te voeren en P' uit P overeenkomstig.

Als voorbeeld (de nummers van de instructies staan tussen vierkante haken):

$\big($[1] `i = 5;` [2] `j = 1;` [3] `exit(0);` , 1 , {(i,17),(j,-3)}$\big) \rightarrow$
$\big($ [1] `i = 5;` [2] `j = 1;` [3] `exit(0);` , 2 , {(i,5),(j,-3)}$\big)$

Dit herschrijfsysteem kennen jullie en het beschrijft exact de uitvoering van Javaprogramma's.

Soms is de herschrijfregel niet toepasbaar omdat de instructie niet kan uitgevoerd worden: bijvoorbeeld als in x = y + 1, y nog niet geïnitialiseerd was en dus niet in *varassoc* voorkwam. Dat is dus het einde van het programma, maar we beschouwen het als een abnormaal einde. Een normaal einde is wanneer P wijst naar de *exit(0)* instructie. Een drietal $(prog, P', varassoc')$ is een minimaal element voor een initiële term $(prog, P, varassoc)$ indien P' wijst naar *exit(0)* en ook

$$(prog, P, varassoc) \rightarrow \ldots \rightarrow (prog, P', varassoc')$$

Gewoonlijk zijn we enkel geïnteresseerd in initiële termen waarvan P wijst naar de eerste instructie van het programma; *varassoc* kan een lege verzameling zijn, maar ook termen met een niet lege *varassoc* zijn zinvol, want op die manier kunnen we het programma van input voorzien.

De vraag *"heeft deze term een minimaal equivalent?"* betekent nu eigenlijk:

> stopt dit programma?

Ondertussen heb je genoeg ervaring met programmeren om te weten dat dit niet altijd duidelijk is!

De vraag *"indien een term een minimaal equivalent heeft, is dat dan uniek?"* klinkt misschien heel gek, omdat het zoveel betekent als

> is het resultaat van een programma uniek bepaald?

en daarop heb je waarschijnlijk blindelings vertrouwd, maar is het niet nuttig van dat zeker te weten? Van het te bewijzen? Echt te begrijpen? Ook het aspect van berekeningskracht van dit herschrijfsysteem - of anders gezegd van Java zonder methoden - kunnen we bestuderen. Of varianten van de uitvoering van Java die toelaten om soms bijvoorbeeld meerdere instructies tegelijk uit te voeren.

De relevantie van herschrijfsystemen voor de informatica is nu wel duidelijk. Nu zou men kunnen beslissen om enkel het Java-herschrijfsysteem te bestuderen omdat dit de taal is waarin we programmeren, maar het is beter om eerst meer

abstracte herschrijfsystemen te bestuderen en daarna het verband met concrete programmeertalen te bekijken. In dit hoofdstuk zullen we als herschrijfsysteem de λ-calculus bestuderen: het is een herschrijfsysteem dat gebaseerd is op het definiëren van functies en de basis is voor functionele programmeertalen als LISP, Scheme, Haskell .... We zullen termen in λ-calculus invoeren en herschrijfregels definiëren die intuïtief redelijk lijken. We zullen eigenschappen bestuderen i.v.m. de uitdrukkingskracht van het formalisme, de uniciteit van eindvormen, eindigheid van herschrijven, strategie van gebruik van reductieregels en het verband met Java. We zullen ook de relevante stellingen bestuderen over confluentie en eindigheid, die maken dat programmeren in het formalisme zinvol is.

Het voordeel van niet Java te bestuderen is dat we duidelijker zullen zien wat strikt nodig is voor een programmeertaal en in het bijzonder dat de problemen met imperatieve talen (waartoe Java behoort) kunnen vermeden worden: neveneffecten (in het bijzonder het gebruik van niet-lokale variabelen en van var parameters) kunnen het moeilijk maken om de betekenis (of bedoeling) van een programma te begrijpen, laat staan nauwkeurig te beschrijven. Het feit dat aan programmavariabelen tijdens een berekening meer dan eens een waarde kan toegekend worden, maakt het ook niet eenvoudiger. Functionele talen, gebaseerd op de λ-calculus, zijn in hun zuivere vorm vrij van die problemen. Logische talen gebaseerd op predicatenlogica eveneens.

2 Λ-calculus

De λ-calculus is een herschrijfsysteem met een bijbehorende theorie over de eigenschappen van dat herschrijfsysteem. In de volgende secties zullen we de termen (λ-expressies genoemd) en de herschrijfregels (of reductieregels) van de λ-calculus invoeren, eerst op informele wijze, daarna meer formeel. Verder zullen we de basisstellingen van Church-Rosser formuleren en bespreken hoe die stellingen aantonen dat programmeren in λ-calculus zinvol is, of op zijn minst dat een programmeertaal gebaseerd op λ-calculus zinvol is. Vooraf nog opmerken dat we enkel niet-getypeerde λ-calculus bekijken; dit wil o.a. zeggen dat functies waarvan je misschien verwacht dat ze enkel op gehele getallen gedefinieerd zijn, ook mogen toegepast worden op strings, boolse waarden of andere functies. Getypeerde λ-calculus is strenger net zoals Java: van elke functie moet een typering gegeven (of mogelijk) zijn.

2.1 De syntax en semantiek van λ-calculus - informeel.

Een paar voorbeelden:

(+ 4 5)

is een eenvoudige uitdrukking (λ-expressie genoemd) in λ-calculus; we kunnen de expressie lezen als:

de functie + toegepast op de argumenten 4 en 5

doch die manier van lezen is eerder een intuïtievormend hulpmiddel dan een noodzaak.

De toepassing van een functie op zijn argumenten wordt in λ-calculus steeds in prefixvorm geschreven. Haakjes zijn er om sommige uitdrukkingen ondubbelzinnig te maken; bijvoorbeeld:

(+ (∗ 5 6) (∗ 8 3))

De buitenste haakjes zijn eigenlijk overbodig en kunnen weggelaten worden.

In een expressie kan men soms een opeenvolgende rij tekens vinden die op zichzelf een λ-expressie vormen; zulk een rij tekens noemen we een *deel-expressie*. Als voorbeeld:

in (+ (∗ 5 6) (∗ 8 3)) vinden we de deel-expressies (∗ 5 6) en (∗ 8 3); ook 5, 6, 8 en 3 zijn deel-expressies; de rij tekens 6) (∗ 8 is geen deel-expressie.

Sommige deel-expressies kunnen herschreven worden (of gereduceerd) m.b.v. de herschrijfregels van de λ-calculus: zulk een deel-expressie heet een *redex*. De

belangrijkste herschrijfregel in de λ-calculus definieert functie-evaluatie: als de deelexpressie een functietoepassing op argumenten is, mag deze herschreven worden volgens de definitie van die functie; daarbij worden de formele parameters vervangen door hun actuele waarde. Net zoals voor elk herschrijfsysteem, zal de semantiek van een λ-expressie gelijk zijn aan zijn *normaalvorm*, t.t.z. een herschreven vorm die geen redex meer bevat.

Het laatste voorbeeld bevat 2 redexen: $(* \ 5 \ 6)$ en $(* \ 8 \ 3)$. De hele expressie is geen redex omdat + enkel reduceerbaar is als er 2 getallen als argumenten staan. De deel-expressies 5,6,8 en 3 zijn geen redexen, want ze zijn niet reduceerbaar. Als we de linkse redex eerst herschrijven of reduceren, krijgen we:

$$(+ \ (* \ 5 \ 6) \ (* \ 8 \ 3)) \rightarrow (+ \ 30 \ (* \ 8 \ 3))$$

waarbij de pijl gelezen wordt als 'reduceert tot'. Nu kan de enige overgebleven redex gereduceerd worden en dat geeft:

$$(+ \ 30 \ (* \ 8 \ 3)) \rightarrow (+ \ 30 \ 24)$$

hetgeen een nieuwe redex creëert die kan gereduceerd worden tot

$$(+ \ 30 \ 24) \rightarrow 54$$

en 54 kan niet verder gereduceerd worden.

We merken hier al op dat als er meer dan één redex is, de volgorde waarin de redexen gereduceerd moeten worden, niet vastligt, t.t.z. er moet een strategie gedefinieerd worden die een volgorde oplegt. Momenteel is er (nog) geen dwingende reden om één strategie te verkiezen boven een andere.

Toepassen van functies en *currying*

De essentie van λ-calculus is functieapplicatie: de volgende notatie wordt gebruikt, waarin f een functie voorstelt en x het (enige) argument waarop f wordt toegepast:

$$f \ x$$

wat we lezen als

f toegepast op het argument x

of nog

f van x

Dit geeft een notatie voor functies met één argument.

De toepassing van een functie met meerdere argumenten wordt beschreven als een opeenvolging van de toepassing van functies met slechts één argument. Dit wordt geïllustreerd door:

$$(+\ 3\ 4) == ((+\ 3)\ 4)$$

In plaats van + te beschouwen als de functie

$$+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

beschouwen we + als een functie:

$$+ : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$$

Dus, + beeldt een geheel getal af op een functie die op zijn beurt een geheel getal afbeeldt op een geheel getal; of nog: $(+\ 3)$ is de functie die elk geheel getal i afbeeldt op $3 + i$.

De veralgemening (tot meerdere argumenten) hiervan ligt voor de hand. In de context van wiskundige logica, werd de methode oorspronkelijk ingevoerd door M. Schönfinkel in 1924, en vervolgens door Haskell B. Curry (1958) zoveel gebruikt dat die methode nu bekend staat als 'currying'[2]. We moeten (voorlopig) achter currying niet meer zoeken dan een manier om de notatie te vereenvoudigen.

Hoeveel haakjes zijn er nodig?

$$((f\ ((+\ 4)\ 3))\ (g\ x))$$

is helemaal ondubbelzinnig, doch met een eenvoudige afspraak worden er heel wat haakjes overbodig:

functieapplicatie is links associatief en werkt op één argument

Daardoor kan de bovenstaande expressie ook geschreven worden als:

$$f\ (+\ 4\ 3)\ (g\ x)$$

Hier volgt een expressie die waarschijnlijk iets anders betekent dan bedoeld werd:

$$+\ +\ 3\ 4\ 5$$

Wat waarschijnlijk bedoeld werd, moet geschreven worden als:

$$+\ (+\ 3\ 4)\ 5$$

Nochtans is er a priori geen enkele reden om de term $+\ +\ 3\ 4\ 5$ als slecht gevormd te beschouwen want ook $(((+\ +)\ 3)\ 4)\ 5$ is goed gevormd. Dit is de eerste kennismaking met het feit dat λ-calculus typeloos is.

[2] geef toe: 'schönfinkeling' had geen kans om populair te worden

Ingebouwde functies en constanten

Zuivere λ-calculus heeft geen ingebouwde functies noch constanten. Voor ons eigen gemak zullen we echter van een aantal functies en constanten veronderstellen dat ze ingebouwd zijn. Dit is echter geen beperking daar men kan aantonen, zoals we van sommige voorbeelden zullen doen, dat ze gedefinieerd kunnen worden in zuivere λ-calculus. Wij veronderstellen dat we beschikken over de volgende ingebouwde constanten:

- de gehele getallen
- TRUE FALSE
- lettertekens zoals 'a' 'b' enzovoort
- NIL (dikwijls gebruikt als lege lijst)

Als ingebouwde functies laten we toe:

- de arithmetische $(* - + /)$
- de logische (AND OR NOT)
- de conditionele (IF)

De constanten kunnen beschouwd worden als functies zonder argument. De andere functies hebben reductieregels, b.v.:

$- \ 4 \ 3 \ \rightarrow \ 1$

AND TRUE FALSE $\rightarrow$ FALSE

IF TRUE $E_t \ E_f \ \rightarrow \ E_t$

IF FALSE $E_t \ E_f \ \rightarrow \ E_f$

Zoals je ziet (bij IF), kan een functie meer dan één reductieregel hebben, doch er is er steeds slechts één van toepassing op een bepaalde uitdrukking: het zijn per slot van rekening functies.

De keuze van de ingebouwde functies is ook tot op zekere hoogte willekeurig en als we er nieuwe nodig hebben, zullen we ze invoeren. Langs de andere kant moeten we opmerken dat de constanten zoals TRUE en FALSE en functies zoals IF, in λ-calculus zelf kunnen gedefinieerd worden (zie sectie 2.5). Ook de gehele getallen zullen we daar voorstellen in zuivere λ-calculus.

Λ-abstracties

De λ-abstractie geeft de mogelijkheid om nieuwe functies te definiëren. Uiteindelijk vallen die definities terug op de ingebouwde functies. Een voorbeeld:

$$(\lambda x.+\ x\ 1)$$

stelt de functie voor die toegepast op een argument, als waarde heeft: dat argument plus 1. Dus

$$(\lambda x.+\ x\ 1)\ 4 \rightarrow 5$$

Let wel: bovenstaande reductie is misschien intuïtief duidelijk, maar tot nu toe niet gedefinieerd.

In een λ-abstractie vind je de volgende elementen terug:

- de λ
- de formele parameter: één variabele
- het . om aan te duiden dat het *lichaam* van de functie begint
- de expressie die aangeeft hoe de functie moet *uitgerekend* worden

Er is een grote gelijkenis met de Java methode:

```
int inc(int x)
{ return(x + 1); }
```

Het belangrijke verschil is dat het lichaam van een Javamethode een rij bevelen bevat die uitgevoerd moeten worden om de terugkeerwaarde te berekenen, terwijl de λ-abstractie een uitdrukking bevat die moet gereduceerd worden om het functieresultaat te kennen.

Om een λ-abstractie ondubbelzinnig te kunnen lezen, moet je weten dat het lichaam van de λ-abstractie zich uitstrekt naar rechts, zover als mogelijk is: je kan haakjes toevoegen om de betekenis volledig te specifiëren indien nodig. Dus

$$(\lambda x.+\ x\ 1)\ 4$$

is hetzelfde als

$$(\lambda x.(+\ x\ 1))\ 4$$

maar niet hetzelfde als

$$(\lambda x.\ +\ x\ 1\ 4)$$

2.2 De syntax van λ-expressies

We beschrijven de syntax van de λ-expressie ($< exp >$) m.b.v. het BNF-formalisme (achter de % staat commentaar):

$$
\begin{array}{lll}
< exp > \implies & < constante > & \% \; ingebouwde \; constanten \\
& | < variable > & \% \; namen \; van \; veranderlijken \\
& | < exp >< exp > & \% \; toepassing \; van \; functie \; op \; argument \\
& | \; \lambda < variable > . < exp > & \% \; \lambda - abstractie \\
& | \; (< exp >)
\end{array}
$$

Veranderlijken zullen we steeds een naam geven die begint met een kleine letter, en hoofdletters gebruiken we voor de afkorting van λ-abstracties, ingebouwde functies en constanten.

Gegevensstructuren?

We willen uiteindelijk aantonen dat we kunnen programmeren in λ-calculus en in andere programmeertalen zijn *gegevensstructuren* of *datastructuren* even belangrijk als de instructies in een programma. Waar zitten nu de gegevensstructuren in λ-calculus? Bedenk dat enerzijds een gegevensstructuur in zekere zin maar bestaat bij de gratie van de functies die erop inwerken en anderzijds dat functies zelf als gegevensstructuur kunnen dienen. Dat zal duidelijker worden in sectie 2.5.

2.3 Terminologie: gebonden en vrije veranderlijken

Beschouw de λ-expressie:

$(\lambda x. \; + \; x \; y)$

Die expressie definieert een functie, maar die functie is pas helemaal gekend als y gegeven is. De waarde van x hebben we niet nodig, t.t.z. x is een formele parameter van de functie en pas bij het toepassen van de functie is er sprake van een waarde aan die parameter te geven. Er is dus een onderscheid tussen x en y. Een andere manier om het verschil tussen x en y te zien is dat intuïtief bovenstaande functie gelijk is aan

$(\lambda z. \; + \; z \; y)$

maar niet aan

$(\lambda x. \; + \; x \; z)$

t.t.z. de voorkomens van x kan je hernoemen zonder de betekenis te veranderen, maar niet dat van y.

Dat onderscheid tussen x en y wordt gevat in de notie van **gebonden of vrij voorkomen van een variabele**. Soms spreekt men van een gebonden of een

vrije variabele, doch het is het voorkomen van een variabele dat vrij is of gebonden en een variabele kan in één expressie zowel vrije als gebonden voorkomens hebben; men zou ook kunnen argumenteren dat het niet om dezelfde variabele gaat. Voorbeelden zullen dat duidelijk maken. We zeggen dat x gebonden voorkomt in bovenstaande λ-expressie, terwijl y er vrij in voorkomt.

Definitie 2.1 Vrij voorkomen van een variabele
Een voorkomen van de variabele x is **vrij** in de expressie E indien

- $E == x$
- $E == (A\ B)$ en dat voorkomen van x is vrij in A of B
- $E == (\lambda y.A)$ en $x \neq y$ en x komt vrij voor in A

Elk voorkomen van een variabele dat niet vrij is, is gebonden. Daaruit kunnen we besluiten dat

- elk voorkomen van de variabele x in E gebonden is in $(\lambda x.E)$
- indien x gebonden voorkomt in A of B, dan is dat voorkomen van x in $(A\ B)$ gebonden

Samengevat: een voorkomen van een variabele is gebonden als er een omsluitende λ-abstractie is die de variabele bindt en anders vrij. Maar pas op: niet alle gebonden voorkomens van een variabele, zijn gebonden door dezelfde λ.

Voorbeelden

In $\lambda x.+((\lambda y.+\ y\ z)\ 7)\ x$
is het voorkomen van x en y gebonden en z vrij.

In $+\ x\ ((\lambda x.+\ x\ 1)\ 4)$
is het meest linkse voorkomen van x vrij, het andere gebonden.

2.4 De semantiek van λ-calculus

Tot hiertoe beschreven we enkel de syntax van λ-calculus formeel; de semantiek werd op een paar voorbeelden verduidelijkt. We zullen nu de semantiek vastleggen. Dit gebeurt door de herschrijfregels van de λ-calculus vast te leggen. Deze herschrijfregels zijn

- α-conversie
- β-conversie
- η-conversie

β-conversie = β-reductie en β-abstractie

Vermits een λ-abstractie een functie beschrijft, moeten we ook aangeven hoe die kan toegepast worden op zijn argument. De volgende herschrijfregel maakt dat duidelijk:

β-reductie

Het resultaat van de toepassing van een λ-abstractie op een argument is een kopie van het lichaam van de λ-abstractie waarin de **vrije** voorkomens van de formele parameter vervangen zijn door (een kopie van) het argument. Merk op: dit is een (syntactische) herschrijfregel.
We zullen verder zien waarom de aandacht getrokken werd op **vrije**.

Voorbeelden:

- $(\lambda x.+\ x\ x)\ 5 \rightarrow +\ 5\ 5$
- $(\lambda x.\ 3)\ 5 \rightarrow 3$
- $(\lambda x.(\lambda y.-\ y\ x))\ 4\ 5 \rightarrow (\lambda y.-\ y\ 4)\ 5 \rightarrow -\ 5\ 4$

In het laatste voorbeeld zie je dat het resultaat van een λ-abstractie terug een λ-abstractie kan zijn: het functieresultaat van een functie kan inderdaad een functie zijn. Omdat currying soms omslachtig is, bestaat er een alternatieve notatie voor $(\lambda x.(\lambda y.E))$ waarin E een λ-expressie is: $(\lambda x\ y.E)$

Functies kunnen ook argument zijn:

$$(\lambda f.f\ 3)\ (\lambda x.+\ x\ 1) \rightarrow (\lambda x.+\ x\ 1)\ 3 \rightarrow +\ 3\ 1$$

Een kopie van het argument (dat hier een λ-abstractie is) komt op elke plaats in de expressie waar de formele parameter f staat.

β-reductie is functie-evaluatie en een implementatie van λ-calculus moet zeker β-reductie efficiënt maken.

Pas op met de namen van variabelen!

$(\lambda x.\ (\lambda x.\ x)\ 7)\ 9 \rightarrow (\lambda x.\ x)\ 7 \rightarrow 7$
Merk op dat de binnenste x niet door 9 werd vervangen omdat dat voorkomen van x door een λ wordt 'beschermd' of gebonden: de definitie van β-reductie zegt duidelijk dat enkel de **vrije** voorkomens van x in E moeten vervangen worden. Indien die binnenste x wel door 9 was vervangen, dan was een verkeerd resultaat bekomen.

β-reduceer zelf correct:

$$(\lambda x.\lambda y.+\ x\ ((\lambda x.-\ x\ 3)\ y))\ 5\ 6$$

(je moet 8 bekomen)

β-abstractie

β-reductie kan ook omgekeerd worden gebruikt:

$$+\ 4\ 1 \rightarrow (\lambda x.+\ x\ 1)\ 4$$

Deze operatie wordt β-abstractie genoemd.

Een andere mogelijkheid om β-reductie omgekeerd te gebruiken, was:

$$+\ 4\ 1 \rightarrow (\lambda x.+\ 4\ x)\ 1$$

Met β-conversie bedoelen we in het vervolg β-reductie of β-abstractie en we noteren het met $\overset{\beta}{\longleftrightarrow}$.

β-conversie drukt een equivalentie uit tussen twee λ-expressies: expressies die er syntactisch verschillend uitzien, maar die we als gelijk willen beschouwen. We zullen nog twee herschrijfregels definiëren.

α-conversie

Het is duidelijk dat $(\lambda x.+\ x\ 1)$ hetzelfde zou moeten betekenen als $(\lambda y.+\ y\ 1)$. Dit wordt uitgedrukt door de hernoemingsregel die **α-conversie** heet en genoteerd wordt met $\overset{\alpha}{\longleftrightarrow}$, dus

$$(\lambda x.+\ x\ 1) \overset{\alpha}{\longleftrightarrow} (\lambda y.+\ y\ 1)$$

We kunnen dus een gebonden variabele hernoemen zolang dat maar systematisch gebeurt, t.t.z. $(\lambda x.E) \overset{\alpha}{\longleftrightarrow} (\lambda y.E')$ indien E' verkregen wordt door in E alle vrije voorkomens van x te vervangen door y. We moeten daar echter ook wat mee opletten:

$$(\lambda x.\ (\lambda y.\ +\ x\ y))$$

willen we toch niet equivalent beschouwen aan

$$(\lambda y.\ (\lambda y.\ +\ y\ y))$$

immers:

$$\begin{array}{lcl}(\lambda x.(\lambda y.+\ x\ y))\ 3\ 4 & \rightarrow & (\lambda y.+3\ y)\ 4 \rightarrow +\ 3\ 4\\ & \rightarrow & 7\end{array}$$

en

$$\begin{array}{lcl}(\lambda y.(\lambda y.+\ y\ y))\ 3\ 4 & \rightarrow & (\lambda y.+\ y\ y)\ 4 \rightarrow +\ 4\ 4\\ & \rightarrow & 8\end{array}$$

Bij α-conversie van $(\lambda x.E)$ mogen we dus x niet vervangen door de naam van een gebonden variabele in E, of er kunnen problemen ontstaan.

η-conversie

η-conversie drukt uit dat bijvoorbeeld $(\lambda x.\ +\ 1\ x)$ equivalent is met de functie $(+\ 1)$. In het algemeen:

$(\lambda x.F\ x) \overset{\eta}{\longleftrightarrow} F$ indien F een functie is en x niet vrij bevat

η-equivalentie steunt op het feit dat linker- en rechterlid toegepast op een argument, equivalent zijn op β-reductie na:

$(\lambda x.F\ x)\ a \overset{\beta}{\longleftrightarrow} F\ a$

en de voorwaarde dat x niet vrij in F mag voorkomen is daarmee duidelijk.

De voorwaarde dat F een functie moet zijn, verhindert bijvoorbeeld dat TRUE η-convertibel is tot $(\lambda x.\text{TRUE}\ x)$.

Het probleem van het vangen van de naam van een variabele

Een voorbeeld: welke van de twee is juist?

$$\begin{array}{lcl}(\lambda x.\ ((\lambda f.\lambda x.\ (f\ x))\ x))\ z & \rightarrow & (\lambda x.\ (\lambda x.\ (x\ x)))\ z\\ & \rightarrow & (\lambda x.\ (x\ x))\end{array}$$

of

$$\begin{aligned}(\lambda x.\ ((\lambda f.\lambda x.\ (f\ x))\ x))\ z &\rightarrow (\lambda a.\ ((\lambda f.\lambda x.\ (f\ x))\ a))\ z \\ &\rightarrow (\lambda a.\ (\lambda x.\ (a\ x)))\ z \\ &\rightarrow \lambda x.\ (z\ x)\end{aligned}$$

Met de eerste reductie is iets raars aan de hand: het argument x wordt na de β-reductie plots een 'formele' parameter. De naam x wordt gevangen in het bereik van de λx; x wordt van vrij ineens gebonden. Dit probleem heet het 'name capture problem' en het is duidelijk dat de β-reductie niet mag uitgevoerd worden als daardoor een vrije veranderlijke gevangen wordt. Het toont ook de noodzaak van α-conversie aan.

Een samenvatting van de conversieregels

We hebben tot nu toe 3 reductieregels gezien, waarmee we λ-expressies kunnen reduceren:

1. naamverandering: α-reductie; de naam van een formele parameter van een λ-abstractie mag overal vervangen worden door een andere naam - let op de restricties; notatie: $\xrightarrow{\alpha}$
2. functietoepassing: β-reductie; een λ-abstractie kan op een argument toegepast worden door een kopie van zijn lichaam te maken en het argument te substitueren voor de vrije voorkomens van de formele parameter; men moet daarbij oppassen indien het argument zelf vrije veranderlijken bevat; notatie: $\xrightarrow{\beta}$
3. eliminatie van redundante λ-abstracties: η-reductie; notatie: $\xrightarrow{\eta}$

Soms wordt het toepassen van ingebouwde functies als een vorm van reductie beschouwd: δ-reductie.

α-reductie is symmetrisch, doch β-reductie en η-reductie niet. β-abstractie (omgekeerde β-reductie) noteren we door $\xleftarrow{\beta}$; omgekeerde η-reductie door $\xleftarrow{\eta}$. We zullen meestal gewoon $\longleftarrow$, $\longrightarrow$ en $\longleftrightarrow$ schrijven als door de context duidelijk is welke regel gebruikt wordt.

Een rij reducties wordt aangeduid door $\xleftrightarrow{*}$ (als het om conversies gaat) of $\xrightarrow{*}$ (voor reducties).

2.5 Programmeren in λ-calculus

Na de invoering van Turingmachines, hebben we laten zien op welke manier getallen kunnen voorgesteld worden en hoe we ermee kunnen rekenen. Ook voor λ-calculus geldt de These van Church en in dit hoofdstuk zullen we laten zien dat deze bewering niet uit de lucht gegrepen is. In het bijzonder zullen we laten zien hoe de natuurlijke getallen kunnen voorgesteld worden in λ-calculus en hoe ermee gerekend kan worden. Bovendien laten we zien hoe een aantal ingebouwde functies toch in zuivere λ-calculus kunnen gedefinieerd worden. Dat gebeurt in de volgende secties. Uiteindelijk zullen we laten zien dat ook *recursieve* functies in λ-calculus kunnen geschreven worden: zie 2.7.

De natuurlijke getallen in λ-calculus

We definiëren eerst $F^n(M)$ voor $n \in \mathbb{N}$ op een inductieve manier:

- $F^0(M) = M$
- $F^{(n+1)}(M) = F(F^n(M))$

Vervolgens definiëren we de *Church numerals* $c_0, c_1, \ldots$:

$$c_n = \lambda f x.\ f^n(x)$$

Deze functies (in twee veranderlijken) stellen de natuurlijke getallen voor: c_0 is de voorstelling van 0, c_1 van 1 enzovoort. Die bewering krijgt meer kracht als we bijvoorbeeld optelling konden definiëren. Rosser toonde hoe dat kan; hij definieerde:

- $A_+ = \lambda xypq.x\ p\ (y\ p\ q)$
- $A_* = \lambda xyz.x\ (y\ z)$
- $A_{\exp} = \lambda xy.y\ x$

Stelling 2.2 $\forall n, m \in N$:

1. $A_+ c_n c_m = c_{n+m}$
2. $A_* c_n c_m = c_{n*m}$
3. $A_{\exp} c_n c_m = c_{n^m}$ (voor $m > 0$)

Bewijs Het bewijs kan gebeuren door inductie maar is op zichzelf niet interessant. ■

Zie ook de oefeningen.

TRUE, FALSE en IF

Hier volgen definities voor TRUE, FALSE en IF:

- TRUE $= \lambda x\ y.x$
- FALSE $= \lambda x\ y.y$
- IF $= \lambda x\ y\ z.(x\ y\ z)$

In de oefeningen wordt gevraagd om aan te tonen dat
IF TRUE $Et\ Ef \rightarrow Et$ en IF FALSE $Et\ Ef \rightarrow Ef$.

Wat gebeurt er als we IF toepassen op een eerste argument dat niet TRUE of FALSE is? Reken eens uit: IF+ 3 4 (je zou 7 moeten bekomen). Dit voorbeeld laat zien dat de functie IF zich enkel als een if-then-else gedraagt, indien toegepast op een boolean.

Merk op dat we niet beweerden dat bovenstaande definities **de** definities van TRUE, FALSE en IF zijn: inderdaad, andere definities zijn mogelijk, doch het is steeds de combinatie van één definitie voor IF met één stel definities voor TRUE en FALSE, die voldoet aan de relatie die we verwachten tussen de drie functies.

HEAD, TAIL en CONS

Als we in Java rijen behandelen met gelinkte lijsten, dan kunnen we het eerste element van de lijst nemen, ook wel de kop genoemd of de head. De rest van de lijst zonder de kop, is de staart of tail. En als we een element hebben en een lijst, dan kunnen we een nieuwe lijst maken met dat eerste element als eerste en de lijst als staart: deze operatie is een concatenatie of cons. Die drie operaties, neem_kop, neem_staart en maak_nieuw, voldoen aan de invarianten: voor elke lijst l en element e

- $neem\_kop(maak\_nieuw(e, l)) = e$
- $neem\_staart(maak\_nieuw(e, l)) = l$

In feite zijn het deze invarianten die bepalen wat een lijst is, of m.a.w. elke drie functies die aan bovenstaande invarianten voldoen, definiëren de lijst, of een type lijst.

In Java zijn we gewoon van over lijsten te denken als datastructuur, maar het is even goed mogelijk - en soms zelfs nuttiger - om over lijsten te denken als een implementatie van de drie functies hierboven. In λ-calculus hebben we niet direct datastructuren, maar wel functies en we kunnen dus de vraag stellen of het mogelijk is in λ-calculus drie functies te definiëren die lijsten bepalen. Dat gaat als volgt:

Definieer

- CONS = $(\lambda a.\lambda b.\lambda f.f\ a\ b)$
- HEAD = $(\lambda c.c(\lambda a.\lambda b.a))$
- TAIL = $(\lambda c.c(\lambda a.\lambda b.b))$

Laten we eens nagaan of we kunnen bewijzen dat

$$\text{HEAD (CONS } p\ q) \ \rightarrow\ p$$

en

$$\text{TAIL (CONS } p\ q) \ \rightarrow\ q$$

Inderdaad

$$\begin{array}{lcl}
\text{HEAD (CONS } p\ q) & = & (\lambda c.c\ (\lambda a.\ \lambda b.\ a))\ (\text{CONS } p\ q) \\
& \rightarrow & \text{CONS } p\ q\ (\lambda a.\ \lambda b.\ a) \\
& = & (\lambda a.\ \lambda b.\ \lambda f.\ f\ a\ b)\ p\ q\ (\lambda a.\ \lambda b.\ a) \\
& \rightarrow & (\lambda b.\ \lambda\ f.f\ p\ b)\ q\ (\lambda a.\ \lambda b.\ a) \\
& \rightarrow & (\lambda f.\ f\ p\ q)\ (\lambda a.\ \lambda b.\ a) \\
& \rightarrow & (\lambda a.\ \lambda\ b.\ a)\ p\ q \\
& \rightarrow & (\lambda\ b.\ p)\ q \\
& \rightarrow & p
\end{array}$$

en

$$\begin{array}{lcl}
\text{TAIL (CONS } p\ q) & = & (\lambda c.c\ (\lambda a.\lambda b.\ b))\ (\text{CONS } p\ q) \\
& \rightarrow & \text{CONS } p\ q\ (\lambda a.\lambda b.b) \\
& = & (\lambda a.\lambda b.\lambda f.f\ a\ b)\ p\ q(\lambda a.\lambda b.b) \\
& \rightarrow & (\lambda a.\lambda b.b)\ p\ q \\
& \rightarrow & (\lambda b.b)\ q \\
& \rightarrow & q
\end{array}$$

Het blijkt dus dat de ingebouwde lijst-constructor en -selectoren in zuivere λ-calculus kunnen gedefinieerd worden. Meer nog: ALLE ingebouwde functies kunnen gemodelleerd worden als λ-abstracties. Dit is natuurlijk zeer bevredigend vanuit theoretisch standpunt, doch omwille van de efficiëntie hebben implementaties van λ-calculus altijd ingebouwde functies.

2.6 Reductiestrategieën

Indien een λ-expressie geen enkele redex meer bevat, zeggen we dat die expressie in normaalvorm staat en dat de evaluatie volledig is. De evaluatie van een expressie gebeurt door herhaaldelijk een redex te reduceren totdat de expressie in normaalvorm staat. Daar een expressie meer dan één redex kan bevatten, kan evaluatie geschieden langs verschillende paden: neem bijvoorbeeld de expressie:

$$\begin{array}{lcl}(+\ (*\ 3\ 4)\ (*\ 7\ 8)) & \rightarrow & (+\ 12\ (*\ 7\ 8)) \\ & \rightarrow & (+\ 12\ 56) \\ & \rightarrow & 68\end{array}$$

of

$$\begin{array}{lcl}(+\ (*\ 3\ 4)\ (*\ 7\ 8)) & \rightarrow & (+\ (*\ 3\ 4)\ 56) \\ & \rightarrow & (+\ 12\ 56) \\ & \rightarrow & 68\end{array}$$

Hier lijkt het alsof gelijk welke reductieorde eindigt en hetzelfde eindresultaat heeft.

Niet elke expressie heeft een normaalvorm; beschouw bijvoorbeeld de expressie:

$$(\lambda x.\ x\ x)(\lambda x.\ x\ x)$$

die expressie bevat één redex en door één β-reductie uit te voeren verkrijgen we:

$$(\lambda x.\ x\ x)(\lambda x.\ x\ x)$$

t.t.z. terug dezelfde expressie! Bovenstaande expressie is de λ-calculus-equivalent van een oneindige lus.

Bovendien is het mogelijk dat één rij reducties een normaalvorm bereikt en een andere dat niet doet! Neem b.v.:

$$(\lambda x.\ 3)\ ((\lambda x.\ x\ x)\ (\lambda x.\ x\ x))$$

door steeds de meest rechtse redex te kiezen geraken we in een oneindige lus; door de linkse te nemen vinden we in één reductie 3.

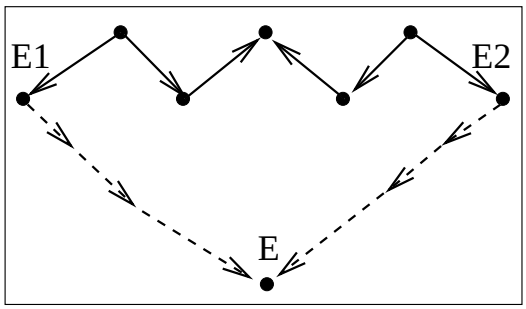
Reductie in normaalorde

Bovenstaande voorbeelden doen een vervelende vraag rijzen: kunnen twee verschillende reductierijen aanleiding geven tot twee verschillende normaalvormen? Indien dit mogelijk is, dan zit λ-calculus in de problemen als model voor functionele berekeningen. Gelukkig is het antwoord 'nee' en dit antwoord is het gevolg van de twee stellingen van Church-Rosser:

Stelling 2.3 Church-Rosser I (CR1)
Indien $E_1 \overset{*}{\longleftrightarrow} E_2$, dan bestaat er een expressie E zodanig dat $E_1 \overset{*}{\rightarrow} E$ en $E_2 \overset{*}{\rightarrow} E$

Bewijs Het bewijs van deze stelling valt buiten het bestek van deze cursus. ■

De figuur 4.1 laat de intuïtie van de stelling zien.



Figuur 4.1: Church-Rosser I

Uit CR1 volgt onmiddellijk de:

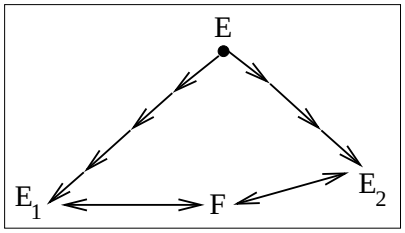
Stelling 2.4 Uniciteit van de normaalvorm
Geen expressie kan geconverteerd worden naar twee verschillende normaalvormen (t.t.z. twee normaalvormen die niet α-convertibel zijn in mekaar).

Bewijs Stel $E \overset{*}{\longleftrightarrow} E_1$ en $E \overset{*}{\longleftrightarrow} E_2$ waarbij E_1 en E_2 in normaalvorm staan, dan is $E_1 \overset{*}{\longleftrightarrow} E_2$ en dus bestaat er volgens CR1 een expressie F zodat $E_1 \overset{*}{\longrightarrow} F$ en $E_2 \overset{*}{\longrightarrow} F$. Maar vermits E_1 en E_2 geen redexen bevatten, is $E_1 = F = E_2$ (op α-conversie na). ■

Een parafrase van CR1 is: alle eindige reductierijen (vanaf een expressie E) eindigen met dezelfde normaalvorm.

Hierbij hoort figuur 4.2.

De tweede stelling van Church-Rosser gaat over een bepaalde reductieorde: de normaalorde.



Figuur 4.2: Gevolg van Church-Rosser I

Definitie 2.5 Reductie in normaalorde
De **normaalorde** bepaalt dat de meest linkse, buitenste redex eerst moet gereduceerd worden. (in het engels: left-most outer-most redex).

Opdat deze definitie zin zou hebben, is het nodig te verstaan dat van twee redexen er steeds één links van de andere staat, ofwel de ene zich binnen de andere bevindt.

Stelling 2.6 Church-Rosser II (CR2)
Indien $E \xrightarrow{*} N$ en N staat in normaalvorm, dan bestaat er een reductierij in normaalorde van E naar N.

Bewijs Ook deze stelling bewijzen we niet. ■

CR1 en CR2 samen geven zoveel als we redelijkerwijze mogen verhopen in een calculus die alle Turing-berekenbare functies kan voorstellen: er is hoogstens één mogelijk resultaat (de redex) van een functietoepassing en reductie in normaalorde vindt het resultaat als het bestaat. Geen enkele reductieorde kan een verkeerd resultaat geven: het enige dat kan misgaan is dat een bepaalde orde niet eindigt want normaalorde garandeert geen eindigheid! Ook dit moesten we verwachten: vermits het halting-probleem geldt voor Turingmachines, moet het ook gelden voor de (Turing)equivalente λ-calculus.

Normaalorde bepaalt dat in:

$$(\lambda x.\ 3)\ ((\lambda y.\ y\ y)\ (\lambda z.\ z\ z))$$

eerst de λx redex gereduceerd moet worden.

Deze orde leunt aan bij de intuïtie dat een argument pas geëvalueerd wordt wanneer het argument gebruikt wordt door de functie: meer naar rechts en meer naar

binnen, staan immers enkel redexen die argument zijn in een grotere expressie die misschien nog kan gereduceerd worden, waardoor dat argument niet hoeft gereduceerd te worden. Dit is een manier van parameters doorgeven die jullie nog niet kennen: bij call-by-value wordt de waarde van de actuele parameter berekend vóór de functie wordt opgeroepen; normaalorde correspondeert met het oproepen van de functie, voordat de actuele parameters berekend worden. Iets gelijkaardig kom je ook tegen in het uitwerken van wiskundige formules; neem bijvoorbeeld de vereenvoudiging van de uitdrukking:

$$\exp(\ln(\cos^2(\Pi/2 + \Pi/6))) + \sin^2(\Pi/2 + \Pi/6)$$

Eén bepaalde volgorde van uitrekenen zou zijn: eerst het argument van cos, dan het kwadrateren van de cos, dan er de ln op toepassen en zo verder. Dat is volgens call-by-value. Een andere volgorde werkt van buiten naar binnen: pas je exp toe op een ln, dan vallen die tegen elkaar weg; pas dan de herschrijfregel

$$\cos^2(x) + \sin^2(x) \longrightarrow 1$$

toe en je hebt het resultaat met heel wat minder moeite.

Een optimale reductieorde?

De normaalorde garandeert dat een normaalvorm gevonden wordt als die bestaat, doch er is geen garantie dat de normaalorde het laagst aantal reductiestappen geeft. Een eenvoudig voorbeeld laat dat zien: normaalordereductie geeft

$$\begin{array}{lcl} (\lambda x.\ +\ x\ x)\ (*\ 3\ 2) & \rightarrow & (+\ (*\ 3\ 2)\ (*\ 3\ 2)) \\ & \rightarrow & (+\ 6\ (*\ 3\ 2)) \\ & \rightarrow & (+\ 6\ 6) \\ & \rightarrow & 12 \end{array}$$

terwijl *call-by-value* geeft

$$\begin{array}{lcl} (\lambda x.\ +\ x\ x)\ (*\ 3\ 2) & \rightarrow & (\lambda x.\ +\ x\ x)\ 6 \\ & \rightarrow & (+\ 6\ 6) \\ & \rightarrow & 12 \end{array}$$

Optimaliteit kan maar bereikt worden als een goede orde samengaat met het vermijden van het meer dan eens uitrekenen van dezelfde expressie. Dit is niet triviaal en op dit ogenblik implementeert geen enkele taal gebaseerd op λ-calculus dat volledig. Bijgevolg kiest een functionele taal een reductieorde waarmee de programmeur moet rekening houden en die de programmeur dan kan proberen uit te buiten.

2.7 Recursieve functies

We hebben al opgemerkt dat λ-calculus elke effectief berekenbare functie kan beschrijven. De meeste interessante functies zijn echter recursief gedefinieerd (in Java, of met een recursievergelijking): zulk een recursieve definitie steunt op de mogelijkheid om de naam van een functie te kunnen gebruiken in zijn definitie. Nu hebben functies in λ-calculus geen naam en λ-calculus lijkt dus recursiviteit te missen. In dit hoofdstuk zullen we laten zien dat recursieve functies kunnen uitgedrukt worden in de λ-calculus zoals tot nu toe beschreven.

Een voorbeeld: de faculteitsfunctie

Beschouw de volgende recursieve definitie van de functie *faculteit*:

$$\text{FAC} = (\lambda n.\ \text{IF}\ (=\ n\ 0)\ 1\ (*\ n\ (\text{FAC}\ (-\ n\ 1))))$$

Deze definitie steunt op de mogelijkheid om λ-abstracties een naam te geven en die naam terug te gebruiken binnen in de λ-abstractie zelf. Λ-calculus heeft zulk een mogelijkheid niet: λ-abstracties zijn naamloze functies en kunnen dus niet naar zichzelf verwijzen. Laten we de definitie van FAC herschrijven als volgt:

$$\text{FAC} = (\lambda f.\ (\lambda n.\ \text{IF}\ (=\ n\ 0)\ 1\ (*\ n\ (f\ (-\ n\ 1))))))\ \text{FAC}$$

Als we $(\lambda f.\ (\lambda n.\ \text{IF}\ (=\ n\ 0)\ 1\ (*\ n\ (f\ (-\ n\ 1))))))$ afkorten tot de functie H, bekomen we

$$\text{FAC} = H\ \text{FAC}$$

en de definitie van H is relatief eenvoudig en in ieder geval geschreven binnen het formalisme van de zuivere λ-calculus.

Merk op het verschil tussen de *afkorting* H en de *naam* FAC: als H ergens in een expressie voorkomt, kan ik H gewoon vervangen door zijn lange vorm en daarmee is H verdwenen uit de expressie. Met FAC gaat dat niet: FAC vervangen door H FAC levert terug een voorkomen van FAC op!

We kunnen de gelijkheid FAC $= H$ FAC zien als een definiërende vergelijking voor FAC: FAC is een functie die voldoet aan FAC $= H$ FAC, of FAC is een vastpunt voor H (want H beeldt FAC op zichzelf af). In het algemeen kunnen functies meer dan één vastpunt hebben: b.v. de functie $(\lambda x.\ *\ x\ x)$ heeft twee vastpunten: 0 en 1. Veronderstel dat we een functie Y kunnen bedenken of construeren die elke functie F afbeeldt op een vastpunt van F, t.t.z. Y voldoet aan

$$\forall F : F\ (Y\ F) = (Y\ F)$$

dus in het bijzonder:

$$H\ (Y\ H) = Y\ H$$

Y wordt een vastpuntcombinator genoemd. Al werken we in niet-getypeerde λ-calculus, we kunnen proberen een *typering* van Y geven: (signatuur is eigenlijk het betere woord)

$$Y : (A \to A) \to A$$

waarin A een typeparameter is; dit drukt uit dat bovenstaande signatuur een typering is voor een willekeurig type A. Zo zou je bijvoorbeeld Y kunnen toepassen op een functie van $\mathbb{N} \to \mathbb{N}$ en als resultaat verwacht je dan iets uit $\mathbb{N}$. Dat werkt niet echt natuurlijk. In feite (omdat we met een niet-getypeerde calculus werken) kan A enkel de verzameling van alle λ-termen zijn.

De zaken even op een rijtje: als we Y in λ-calculus kunnen definiëren, dan hebben we FAC, want dan volstaat het $Y\ H$ uit te rekenen; dit is nu een expressie in zuivere λ-calculus.

Om dus FAC in λ-calculus te definiëren hebben we enkel nog de λ-calculus definitie van Y nodig, want die van H hebben we al. Laten we eerst nog zien dat het genoeg is dat de eigenschap $Y\ H = H\ (Y\ H)$ genoeg is om FAC te hebben; laat ons daartoe eens FAC 1 uitrekenen:

$$\begin{array}{lcl}
\text{FAC } 1 & = & Y\ H\ 1 \\
& = & H\ (Y\ H)\ 1 \\
& = & (\lambda f.\ \lambda n.\ IF\ (=\ n\ 0)\ 1\ (*\ n\ (f\ (-\ n\ 1))))\ (Y\ H)\ 1 \\
& \to & (\lambda n.\ IF\ (=\ n\ 0)\ 1\ (*\ n\ (Y\ H\ (-\ n\ 1))))\ 1 \\
& \to & IF\ (=\ 1\ 0)\ 1\ (*\ 1\ (Y\ H\ (-\ 1\ 1))) \\
& \to & \text{IF FALSE } 1\ (*\ 1\ (Y\ H\ (-\ 1\ 1))) \\
& \to & *\ 1\ (Y\ H\ (-\ 1\ 1)) \\
& \to & *\ 1\ (Y\ H\ 0) \\
& = & *\ 1\ (H\ (Y\ H)\ 0) \\
& = & *\ 1\ ((\lambda f.\ \lambda n.\ \text{IF}\ (=\ n\ 0)\ 1\ (*\ n\ (f\ (-\ n\ 1))))\ (Y\ H)\ 0) \\
& \to & *\ 1\ (\lambda n.\ IF\ (=\ n\ 0)\ 1\ (*\ n\ ((Y\ H)\ (-\ n\ 1)))\ 0) \\
& \to & *\ 1\ (\text{IF}\ (=\ 0\ 0)\ 1\ (*\ 0\ ((Y\ H)\ (-\ 0\ 1)))) \\
& \to & *\ 1\ (\text{IF TRUE } 1\ (*\ 0\ ((Y\ H)\ (-\ 0\ 1)))) \\
& \to & *\ 1\ 1 \\
& \to & 1
\end{array}$$

De definitie van Y in λ-calculus

De gelijkheid $Y\ H = H\ (Y\ H)$ drukt recursiviteit uit in zijn meest elementaire vorm: Y kan gebruikt worden om *alle* recursieve functies mee te definiëren. Nu komt er een stukje λ-magie: we gaan Y definiëren in λ-calculus en de definitie is niet eens zo lang:

$$Y = (\lambda h.\ (\lambda x.\ h\ (x\ x))\ (\lambda y.\ h\ (y\ y)))$$

Laten we nagaan dat $Y\ F = F\ (Y\ F)$ voor een willekeurige F:

$$\begin{array}{rcl} Y\ F & = & (\lambda h.\ (\lambda x.\ h\ (x\ x))\ (\lambda y.\ h\ (y\ y)))\ F \\ & \rightarrow & (\lambda x.\ F\ (x\ x))\ (\lambda y.\ F\ (y\ y)) \\ & \rightarrow & F\ ((\lambda y.\ F\ (y\ y))\ (\lambda y.\ F\ (y\ y))) \\ & \rightarrow & F\ (Y\ F) \end{array}$$

Implementaties van λ-calculus laten meestal recursieve definities toe door een functie syntactisch naar zichzelf te laten verwijzen, zoals bij onze eerste definitie van FAC: de functie Y is een te inefficiënte omweg.

2.8 Programmeren en λ-calculus

Λ-calculus samen met een reductiestrategie (niet noodzakelijk de normaalordestrategie) vormt een programmeertaal: als de reductiestrategie vastligt, dan heeft de programmeur immers houvast om over uitvoering en complexiteit (en eventueel efficiëntie) te redeneren. De normaalordestrategie is speciaal wegens Church-Rosser II en correspondeert met de idee: reken een actuele parameter pas uit als je hem echt nodig hebt. Dit komt neer op een parameterbindingsmechnisme dat *lui* genoemd wordt - ook wel call-by-need. Het staat in contrast met de value-parameter van Java die helemaal geëvalueerd moet worden voor de methode wordt binnengegaan: dat kan eventueel verloren werk zijn, indien de formele parameter (in het lichaam van de methode) niet gebruikt wordt. Maar als de methode echt afhangt van alle argumenten, is het niet onredelijk van de waarde van de actuele parameters van te voren te berekenen. Een functie die van al zijn parameters afhangt wordt *strikt* genoemd (er is een meer formele definitie die hier niet belangrijk is). Naar analogie wordt een strategie die parameters reduceert voor de functie toe te passen, *strikt* genoemd en een strategie die uitstelt van parameters te reduceren *lui*. De normaalordestrategie is dus lui. Een programmeertaal wordt ook op die manier gekarakteriseerd: lui of strikt (in het engels lazy of strict), maar er zijn ook mengvormen.

Java kan je dus beschouwen als een strikte taal. Nochtans is er minstens één constructie in Java lui: de if-then-else. Herinner je dat IF in λ-calculus een functie is. We kunnen dus ook in Java de if-then-else constructie bekijken als een methode met drie argumenten: de test, de reeks instructies uit te voeren als de test waar is (de then-tak), en de reeks instructies uit te voeren als de test onwaar is (de else-tak). Hier een voorbeeld:

if (x = 0) then y = 0 else y = 1/x ;

of in functie-notatie:

y = if-then-else(x=0, 0, 1/x)

Veronderstel dat de functie if-then-else strikt wordt uitgevoerd, dan wordt de waarde van (x=0) bepaald, de waarde van de then-tak (0) en de waarde van de else-tak (1/x): het laatste geeft een deling door 0 indien (x=0)! Daaruit besluiten we dat if-then-else wel strikt is in zijn eerste argument, maar lui moet zijn in het tweede en het derde.

Een ander verschilpunt tussen Java en λ-calculus, is dat in Java geen methoden zijn toegelaten als parameter (toch niet in volle algemeenheid) of methoderesultaat. Een taal die dat wel toelaat is van een *hogere orde*.

μ-Java en λ-calculus

Tot slot gaan we informeel wat dieper in op het verband tussen Java en λ-calculus. Beschouw μ-Java, t.t.z. de deelverzameling van Java die enkel methoden heeft en de if-then-else constructie, maar zonder toekenningen aan veranderlijken [3]. Het feit dat we geen toekenningen hebben, sluit ook het for-statement uit en maakt het gebruik van de while eveneens zinloos. Neveneffecten (t.t.z. toekenningen aan "globale" veranderlijken) bestaan ineens niet meer. Toch kunnen we nog altijd interessante functies schrijven in μ-Java, bijvoorbeeld:

```
int  fib(int i)
{
  if (i < 2) return(1);
  return(fib(i-1) + fib(i-2));
}
```

We hebben nu iets dat erg op λ-calculus gelijkt, doch we missen ook iets: Java laat geen methoden toe als parameter, noch als methoderesultaat. Dat lijkt op het eerste zicht geen groot verlies, maar in λ-calculus maakten we juist gebruik van functies als parameter of resultaat bij het definiëren van HEAD en dergelijke. Hier zie je duidelijk een ruil die Java doet: expliciete datastructuren (klassen met daarin verwijzingen naar objecten) in Java tegen de hogere-orde mogelijkheden van λ-calculus.

[3] we kunnen unieke toekenning eventueel toelaten, maar geen enkele veranderlijke mag meer dan eens een waarde krijgen

3 Andere herschrijfsystemen

We hebben verschillende formalismen om te berekenen bekeken: push-down automaten, Turingmachines, λ-calculus, recursieve functies, Postsystemen ... En je kent ook het gebruik van logica (formules worden herschreven tot een opeenvolging van herschrijvingen een bewijs vormt omdat de laatste stap tot een tautologie of aximo leidt), of in de vorm van een declaratieve taal Horn clauses (Prolog). En dan zijn er de programmeertalen die vanuit theoretisch standpunt wat *messy* zijn, maar vanuit praktisch standpunt beter voldoen.

Ondertussen zie je natuurlijk in dat elk van die rekenmethoden kan beschreven worden door een herschrijfsysteem. Je hebt in een andere cursus CHR geleerd: daar is elke programmaregel een herschrijfregel. Dikwijls kan heel de semantiek van een (programmeer)taal vastgelegd worden door een herschrijfsemantiek. Afhankelijk van welke notatie je kiest ben je dan bezig met denotationele semantiek, of evoluerende algebras, of ...

Maar we mogen niet de fout maken om te denken dat dit alles is: in een aantal formalismen (logica's) is semantiek gebaseerd op modellen. Dat is een meer statisch zicht op semantiek, maar het verband tussen modelsemantiek en een herschrijfsemantiek bestaat altijd, of met andere woorden: waarheid en bewijsbaarheid zijn verbonden met elkaar, maar daarom niet altijd aan elkaar gelijk.

4 Referenties

- Simon L. Peyton-Jones "The implementation of Functional Programming Languages"
- H.P. Barendregt "The Lambda Calculus, its syntax and semantics" (North-Holland, Amsterdam 1984)
- "Handbook of Theoretical Computer Science, volume B: Formal Models and Semantics" (Jan Van Leeuwen, editor)

5 Oefeningen

1. Waarom is het volgende niet waar:

   $$(\lambda x. + xx) \overset{\eta}{\longleftrightarrow} (+x)$$

2. Reken uit:

   $$(\lambda x.x\ 17)\ (+\ 3)$$
   $$(\lambda x.x\ 17)\ (\lambda y.\ 12)$$

3. Welke voorkomens van de variable x zijn vrij in de volgende uitdrukkingen en welke zijn gebonden. Verklaar ook je antwoord heel formeel!

   (a) $\lambda x.(xy)$
   (b) $\lambda x.xxx$
   (c) $(\lambda x.x)(xx)$

4. Duid in de volgende uitdrukking aan welke voorkomens van de variabele x gebonden zijn en geef ook aan door welke λ-abstractie ze gebonden worden.

   $$+\ x\ (\lambda\ x\ .\ (\lambda\ y.\ \lambda\ x.\ +\ (+\ x\ y)\ x\ )x)\ (\lambda\ y.\ +\ x\ y)$$

5. Toon aan dat

   (a) $A_+c_0c_n = c_n$
   (b) $A_+c_1c_m = c_{m+1}$

6. Laat zien:

   (a) $A_*c_0c_m = c_0$
   (b) $A_*c_1c_m = c_m$

7. Bewijs dat: $A_+c_2c_2 = c_4$

8. Definieer zelf de functies AND en OR binnen de zuivere λ-calculus.

9. Bewijs dat $Y\ (Y\ Y) = Y\ Y$, waarbij Y de vastpuntscombinator is.

10. Toon aan dat, met de definities op pagina 148, voldaan is aan:

    IF TRUE $Et\ Ef \rightarrow Et$ en IF FALSE $Et\ Ef \rightarrow Ef$

Hoofdstuk 5

Andere rekenparadigma's

Door al dat formeel gedoe met eindige toestandsmachines, push-down automaten en Turing machines ben je misschien wat al te eenzijdig gaan denken over berekeningen. Hopelijk wordt je intuïtie hier nog een snok gegeven. Ter voorbereiding: wist je dat je met wat nageltjes en een elastiekje een convex omhullende kan berekenen? Wist je dat je met zeepsop een minimaal oppervlak tussen een aantal ribben kan bepalen? Wist je dat enkel het 3-deeltjes probleem analytisch is opgelost, maar dat het n-deeltjes probleem voor eindige maar grote n routine-matig door de natuur wordt opgelost? Wist je dat de meest succesvolle manier om te bepalen of een (groot) getal priem is gebaseerd is op stochastische methoden? Wist je dat er polynome algoritmen bestaan voor sommige NP-complete problemen die de optimale oplossing met een kleine constante factor benaderen? Wist je dat met willekeurige precisie sommige NP-complete problemen in polynome tijd kunnen worden opgelost met stochastische, zelfs determinische algoritmen?

Sommige van die vragen lijken te behoren tot een ander wetenschapsdomein (dan informatica, of deze cursus), maar ...

Soms lijkt er een grote afstand tussen wat we berekenen durven noemen en waarvoor oplossingen bestaan, maar anderzijds is er aan de kunst van het berekenen een zeer utilitaire component: als we een bruikbaar resultaat krijgen, dan zijn we tevreden. Uiteraard willen we ook een theoretische verklaring voor waarom een bepaalde methode een bruikbaar resultaat oplevert, maar a priori mag geen enkele methode worden uitgesloten. Dat zou immers niet-wetenschappelijk zijn en wetenschap is wat aan de(ze) universiteit wordt nagestreefd.

De *andere* paradigma's die hier worden voorgesteld zijn nuttig binnen hun context, en dragen zeker bij tot een beter begrip van wat berekenen is, en welk nuttig resultaat men kan verwachten van een veralgemeen begrip van berekenen. Laat je scepsis even varen en sta open voor wat komt. Sommige methoden zullen je

leven misschien meer beïnvloeden dan we nu denken.

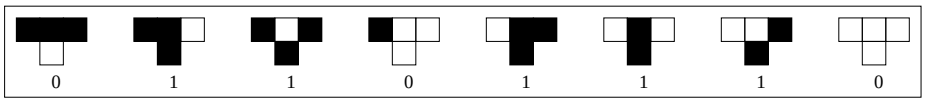
1 Cellulaire automaten

Een **cellulaire automaat** is een verzameling identieke eindige toestandsmachines die allen regelmatig en tegelijk van toestand *veranderen*, en waarvan de toestand afhangt van de toestand van de *buren*. Dikwijls zijn cellulaire automaten gerangschikt volgens een vast patroon - bijvoorbeeld in een 2-dimensionaal rooster - en zijn de buren ook regelmatig, bijvoorbeeld in zulk een 2-dimensionaal rooster de buren links, recht, onder en boven. In termen van onze vertrouwde definitie van eindige toestandsmachine kan zulk een cellulaire automaat beschreven worden door zijn overgangsfunctie δ die als domein nu $Q_{self} \times Q_{left} \times Q_{right} \times Q_{up} \times Q_{down}$ is. Σ kan daar eventueel ook nog bij. Er is dikwijls een bepaalde rusttoestand: als alle machines in die rusttoestand zijn, dan verandert er niks. Je brengt de cellulaire automaat op gang door één machine (of meerdere) in een bepaalde toestand te brengen. Vanaf dan *evolueert* de cellulaire automaat. Men beschouwt zowel eindige als oneindige configuraties van cellulaire automaten.

Voorbeelden van cellulaire automaten:

- Conway's Game of Life: twee-dimensionaal grid; buur is ook diagonaal rakend; er zijn slechts twee toestanden (levend en dood); een dode cel leeft in de volgende cyclus indien die cel exact drie levende buren had; een levende cel sterft indien ie minder dan twee of meer dan drie levende buren heeft, en blijft anders leven. Maak de transisitietabel. Hoe sterk is deze vorm van cellulaire automaten?

- één-dimensionale cellulaire automaten: populair geworden door het book *A New Kind of Science* van Stephen Wolfram; de cellen liggen op een oneindige rij; de interactie met de buren is beperkt tot de twee rakende cellen; er zijn slechts twee toestanden per cel; elke cellulaire automaat van dit type wordt beschreven door zijn transitiefunctie die van het type $\{0,1\} \times \{0,1\} \times \{0,1\} \to \{0,1\}$ is. Je ziet dat er slechts 256 zulke functies zijn, en populair wordt ernaar verwezen door de 8 bits van de range als decimaal getal te vermelden. Zo spreekt men bevoorbeeld van *rule 30* die random getallen kan genereren, en van *rule 110* die *universeel* is. De visuele voorstelling van rule 110 zie je in figuur 5.1.



Figuur 5.1: Rule 110

Zoek uit wat het juist betekent dat een cellulaire automaat universeel is, in het bijzonder wat denk je van het feit dat een (universele) Turing machine output heeft wanneer ie stopt, terwijl een cellulaire automaat nooit stopt? De input voor een berekening door een cellulaire automaat is de initiële toestand van alle cellen: een oneindige array.

Oorsprong: Cellulaire automaten werden het eerst bestudeerd door John von Neumann: hij zocht naar zelfreproducerende eindige toestandsmachines. Zijn design daarvoor bevat een 2-dimensionaal grid van machines met elk 29 toestanden. Dit was baanbrekend werk in een tijd dat DNA (en zijn principes) nog niet ontdekt of bedacht was.

Een klassieker: the firing squad problem .
Beeld je een rij soldaten in, klaar om een veroordeelde te executeren. De generaal die helemaal rechts van de rij staat, kan enkel een bevel geven aan zijn linkerbuur (een gewone soldaat), en elke soldaat kan enkel een bevel doorgeven links en rechts. Hoe kunnen we verkrijgen dat alle soldaten tegelijk vuren? De bedoeling is het te doen met een cellulaire automaat, t.t.z. een eindige toestandsmachine voor elke soldaat, maar wel identiek voor alle soldaten, en eventueel een andere machine voor de generaal.

Dit is een wat lugubere versie van een algemeen synchronisatieprobleem waarbij de communicatie enkel tussen directe buren mogelijk is.

Het probleem komt van John Myhill (die we al kennen van vroeger), en werd eerst opgelost door Marvin Minsky[1] en John McCarthy[2].

Kan je een intuïtieve oplossing voor FSP bedenken? Denk je dat het probleem relevant is? Kan je veralgemeningen bedenken, varianten, ...?

[1] Turing award 1969
[2] De man van LISP, voorloper van Haskell dat gebaseerd is op λ-calculus - Turing award 1971

2 DNA-computing

Deze vorm van *berekenen* werd in 1994 beschreven door Leonard Adleman[3], de **A** in RSA en de uitvinder van de term computervirus. In 2004 commercialiseerde Ehud Shapiro[4] de eerste DNA computer.

Meestal is het gebruikte jargon i.v.m. DNA-computing niet toegankelijk voor niet DNA-ologen en dat is verkeerd: we kunnen het principe ook aanbrengen zonder DNA-terminologie. Wie meer wil leren over de principes kan verder grasduinen natuurlijk.

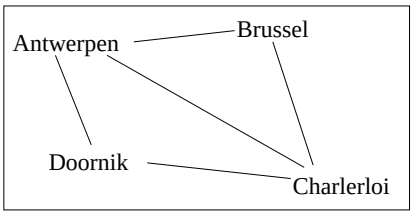
We bekijken hier als voorbeeld hoe het probleem van het berekenen van een Hamiltoniaanse kring (HC) in een graaf met DNA-computing kan worden opgelost. HC is een NP-compleet probleem en ook het eerste waarmee Adleman de pers haalde. Misschien voor een goed begrip: in de klassiekere benaderingen van het probleem, kunnen we gemakkelijk het aantal knopen variëren; het is enkel maar een N in het algoritme. In DNA-computing ligt dat anders: men construeert een fysisch systeem dat toelaat om een bepaalde instance van het algemene probleem op te lossen. Dat accepteren is een stukje van de flexibiliteit t.o.v. berekenen die we je hier vragen. Adleman liet initieel zien hoe het probleem van de Hamiltoniaanse kring kan opgelost worden voor 7 knopen. Wij doen het hier met slechts 4 want dan is het overzichtelijker: de principes zijn natuurlijk toepasbaar op elke grootte.

Het Hamiltoniaanse kring probleem wordt wel eens ingebed in het traveling salesman problem, en daarom noemen we de knopen *steden*. We kiezen 4 steden en gebruiken de eerste letter uit hun naam om ernaar te refereren: Antwerpen, Brussel, Charlerloi en Dinant - A, B, C en D.

Het wegennet dat we voorstellen heeft de vorm als in figuur 5.2. Een weg tussen Antwerpen en Brussel stellen we voor door AB. We hebben wegen AB, AC, AD, BC, CD en omgekeerd. Maak nu dominostenen met daarop AB, AC, AD, BC, CD - **veel** stenen, en voeg er ook maar stenen AA, BB, CC en DD aan toe.
Stel nu dat we aan die dominostenen de volgende eigenschap kunnen geven: drie stenen XY, YZ en YY klitten samen in de vorm als links in figuur 5.3 - andere

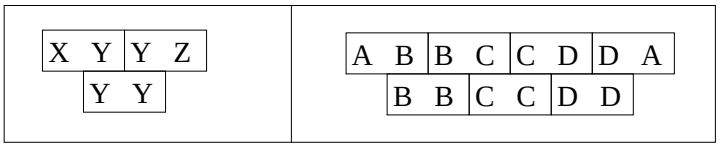
[3] Turing award 2002

[4] Ehud Shapiro doctoreerde over debugging van logische programma's, zeg maar Prolog.



Figuur 5.2: De steden met hun verbindingen

combinaties klitten niet samen.



Figuur 5.3: Links: hoe dominostenen aan elkaar klitten; rechts: een oplossing van HC voor A,B,C,D

Al die stenen van hierboven gooien we nu in een bokaal, we roeren stevig en het samenklitten begint. Een aantal samengeklitte stenen is een cluster. Na een tijdje stopt het samenklitten.

Nu filteren we de clusters:

- clusters die te lang of te kort zijn gaan eruit
- clusters met te weinig verschillende *steden* verdwijnen
- (eventueel) clusters die *beginnen* met de slechte stad ... weg ermee

Wat blijft er nog over:

- niks: dan bestaat er geen HC
- de HC's die beginnen bij een bepaalde stad

Op die manier is het beslissingsprobleem (bestaat een HC [vertrekkend van een bepaalde stad]) constructief opgelost.

Als elk van de operaties hierboven fysisch realiseerbaar is met DNA (of een ander proces) dan hebben we een *algoritme* voor het HC probleem, een algoritme dat geen digitale computer nodig heeft om uitgevoerd te worden.

De kracht van DNA-computing ligt in

- de berekeningsonderdelen zijn inderdaad fysisch realiseerbaar
- massief parallellisme
- miniaturisatie

Wat denk je over deze vorm van berekenen? Heb je vertrouwen in deze manier van berekenen? Denk je dat het iets wordt? Zoek eens op wat E. Shapiro juist op de markt bracht.

3 Ant-computing

Terwijl DNA-computing zijn inspiratie vindt in processen op zeer kleine schaal - chemische processen - is deze manier van berekenen eerder geïnspireerd door de fauna: ook hier hebben we grote aantallen entiteiten nodig om een goed berekeningsresultaat te verkrijgen, entiteiten met weinig individuele berekeningskracht en mieren lijken dat te leveren.[5]

Een mier heeft weinig geheugen en ook weinig globaal overzicht: een beetje zoals FSA's. Verder zijn mieren quasi identiek. Ze lijken dus op cellulaire automaten, maar ze hebben ten opzichte van elkaar geen vaste positie: ze lopen wat rond en wie hun buur is verandert voortdurend. Dus niet echt zoals we verwachten van cellulaire automaten. Toch kan een groep mieren een probleem oplossen, bijvoorbeeld *langs welke (kortste) weg transporteren we voedsel van de bron naar het nest?*. Het oplossen van zulke problemen (zelfs benaderend) is cruciaal voor het voortbestaan van het nest. Hoe doen ze het? Biologen hebben dat bestudeerd en hun bevindingen leidden tot nieuwe manieren om te berekenen:

- mieren vertrekken uit het nest op een randomzoektocht naar voedsel
- ze laten op hun tocht een spoor van feromonen achter: individueel net genoeg om terug te kunnen keren naar hun nest
- een spoor met feromoon is aantrekkelijk om te volgen voor (andere) mieren
- als een mier voedsel vindt, dan keert ie terug naar het nest en laat meer feromoon achter; als ie niks vindt, dan keert ie ook terug, maar laat geen feromoon meer achter; de mier laat ook meer feromoon achter op zijn terugtocht naarmate meer/beter voedsel gevonden werd
- feromoon vervliegt na een tijdje: evaporatie

Dit alles maakt dat

- andere mieren vinden voedsel door de sporen te volgen
- als de voedselbron uitgeput geraakt dan *vergeten* de mieren het pad door die evaporatie

[5] Lees ook eens de conversaties tussen de mierenkolonie en de miereneter in het boek van Hoffstadter *Gödel-Escher-Bach*: het is verhelderend op meerdere vlakken.

- het random effect maakt dat nieuwe voedselbronnen gevonden worden zelfs al is er al een goede voedselbron beschikbaar

De prijs is dat een hele boel mieren sterven bij het random zoeken naar voedselbronnen, of het volgen van onterechte sporen. Maar zolang de kolonie overleeft is dat ok.

De principes van ant-computing zijn o.a. toegepast in de praktijk voor het vinden van kortste paden in een grafe die dynamisch verandert.

Zelf doen:

Geef je oordeel over deze manier van *rekenen*.

Hoe speelt (massief) parallelisme een rol?

Zoek op: stygmergy en de rol ervan in ant-computing.

4 En verder ...

Er is natuurlijk nog veel meer, in volle ontwikkeling of al afgedaan. De universitaire informaticus stelt zich beter open voor de nieuwigheden en kent het oude natuurlijk grondig.

Het is verleidelijk te denken dat *elk van die alternatieve paradigma's kan softwarematig gesimuleerd worden, dus leveren ze niks nieuws.*

Dat is slechts waar voor de eerste helft: simulatie kan, in de zin dat geen enkel paradigma (tot nu toe) de Turing-berekenbaarheid overschrijdt. Maar van quantum computing weet men dat het de complexiteitseigenschappen van beslissingen kan verbeteren. En constante factoren kunnen ook verbeterd worden door massaal parallellisme (zoals in ant- of dna-computing).

Openstaan voor vernieuwing is de boodschap.

Hoofdstuk 6

Talen en Complexiteit

In de vorige hoofdstukken hebben we het volgende probleem bestudeerd: kan een gegeven taal beslist/bepaald worden en welke machinerie is daarvoor nodig. Als een taal kan beslist worden dan bestaat er een algoritme om de beslissing (voor een willekeurige gegeven string) te nemen. De natuurlijke vraag is dan: *hoe duur is het beslissingsproces?* Een natuurlijke maat om die kost in uit te drukken is de grootte van de gegeven string - of die nu behoort tot de taal of niet. De kost is dan een maat voor het aantal elementaire stappen in een algoritme dat het beslissingsproces implementeert. Het is zeker nodig om hier heel precies het kader af te spreken. De kost wordt de complexiteit van de taal genoemd.

Is er een verband tussen de complexiteit waarmee een taal beslist kan worden en zijn plaats in de Chomsky hiërarchie?

Het is gemakkelijk in te zien dat een reguliere taal beslist wordt in $O(n)$ stappen (van de FSA) met n het aantal symbolen in de input. Of dit ook kan in $O(n)$ stappen op een Turing machine valt nog aan te tonen (door jullie). Er is ook een algemeen $O(n^3)$ (op een TM met 3 banden) algoritme voor CFL's: het algoritme van Cocke-Younger-Kasami. Het is een vorm van *chart* of *tabular parsing*.

Voor andere talen is die situatie niet zo eenduidig: de priemgetallen vormen een niet context-vrije taal, maar er is wel een polynomiaal algoritme voor hun beslissing. Voor SAT hebben we dat (nog ?) niet.

Maar deze zaken horen in een andere cursus :-)

Zelf doen: Bewijs dat de priemgetallen niet regulier of contextvrij zijn. Je mag ook een socio-historisch-wetenschappelijk argument geven :-)