Didactische ondersteuning van theoretische informatica

Annelotte BOLLEN

promotor:
Prof. dr. Frank NEVEN

Academiejaar 2004-2005

Eindverhandeling voorgedragen tot het bekomen van de graad
licentiaat in de informatica
afstudeervariant multimedia

Voorwoord

Je studies afronden is het einde van een levensfase. Aan de universiteit gaat dit gepaard met de grootste opgave die je als student moet volbrengen, namelijk het maken van een thesis. Het schrijven van dit voorwoord is een moment om even stil te staan en achteruit te kijken.

Het liep niet steeds van een leien dakje en ik besef eens te meer dat er nog zoveel te ontdekken valt, de opleiding omvat slechts het topje van de ijsberg. Alles leren is namelijk onmogelijk... zeker in een steeds vernieuwende en groeiende opleiding als informatica. Maar door de brede vorming die een universiteit aanbiedt, bezit ik nu de basis die ik nodig heb om te groeien in mijn vak. Tijdens mijn studie heb ik, dankzij de professoren en assistenten, kennis kunnen vergaren van de verschillende deelgebieden in de informatica. In het bijzonder wil ik mijn promotor bedanken voor zijn raad en geduld tijdens het verloop van mijn thesis.

Ik leerde niet alleen over een vakgebied; door de jaren heen heb ik mezelf kunnen ontplooien en leerde ik de deelgebieden van het mens-zijn kennen: taal-, communicatie- en relatievaardigheden. Hiervoor ben ik dank verschuldigd aan al mijn vrienden, in het bijzonder aan Kristien, Johan en Frederik. Zij zorgden mee voor de schitterende tijd die ik in Diepenbeek gehad heb. Het is dankzij de levendige discussies, de lange nachten en de ervaring samen te werken met anderen dat ik geworden ben wie ik nu ben.

Zonder de steun en het geduld van mijn vriend Tom zou ik enkele maanden geleden de handdoek in de ring hebben gegooid. Dankzij hem ben ik echter blijven volhouden en kon ik door het bos de bomen terug zien. Maar niet alleen daarvoor moet ik hem ontzettend bedanken. Ook voor het nalezen van deze thesis en zinvolle ontspanning kon ik steeds op hem rekenen.

En als laatste, maar zeker niet als minste, wil ik graag mijn ouders bedanken. Zij hebben me de mogelijkheid geboden om te studeren, zij blijven onvoorwaardelijk in mij geloven en steunen me in alles wat ik doe. Bedankt!

Annelotte Bollen
10 augustus 2005

Inhoudsopgave

1 Inleiding **5**
- 1.1 Doelstellingen van het vak Theoretische Informatica . . . 5
- 1.2 Methodisch-didactische basisprincipes . . . 6
- 1.3 Indeling van de thesis . . . 7

2 Grammatica's en parsing **8**
- 2.1 Didactische wenken . . . 8
- 2.2 Contextvrije grammatica . . . 9
  - 2.2.1 Voorbeeld . . . 9
- 2.3 Parsing . . . 10
  - 2.3.1 Wat is parsing . . . 10
  - 2.3.2 Het CYK parsing algoritme . . . 10
  - 2.3.3 Voorbeeld . . . 12
- 2.4 CYK parsing in JFLAP . . . 15
  - 2.4.1 Wat is JFLAP? . . . 15
  - 2.4.2 Werkwijze voor CYK parsing in JFLAP . . . 16
- 2.5 Oefeningen . . . 21
  - 2.5.1 Oefening 1 . . . 21
  - 2.5.2 Oefening 2 . . . 21
- 2.6 Besluit . . . 21

3 Stringautomaten en reguliere expressies **22**
- 3.1 Didactische wenken . . . 22
- 3.2 Thompson-constructie . . . 23
  - 3.2.1 Voorbeeld: Thompson-automaat voor $(b^*a(b^*a)^*) + \epsilon$ . . . 24
- 3.3 Glushkov-constructie . . . 26
  - 3.3.1 Voorbeeld: Glushkov-automaat voor $(b^*a(b^*a)^*) + \epsilon$ . . . 27
  - 3.3.2 Expliciete constructie van Glushkov-automaat voor $E = (b^*a(b^*a)^*) + \epsilon$ 29
  - 3.3.3 Eigenschappen van de Glushkov-automaat . . . 30
- 3.4 Vergelijking . . . 31
  - 3.4.1 Van Thompson naar Glushkov . . . 31
  - 3.4.2 Van Glushkov naar Thompson . . . 32
  - 3.4.3 Gebruik van de twee constructies . . . 33
- 3.5 Besluit . . . 33

4 Boomautomaten **34**
4.1 Didactische wenken . . . 34
4.1.1 Differentiatie . . . 34
4.1.2 Integratie . . . 34
4.2 Boomautomaten . . . 35
4.2.1 Boomautomaat en stringautomaat . . . 35
4.2.2 Definitie . . . 35
4.2.3 Voorbeeld: booleaanse formules [13] . . . 36
4.3 Toepassing XML en DTD . . . 37
4.3.1 XML: eXtensible Markup Language . . . 37
4.3.2 Equivalentie van XML . . . 39
4.3.3 DTD's . . . 41
4.3.4 DTD als boomautomaat . . . 44
4.3.5 Ondubbelzinnig gedefinieerde DTD's . . . 46
4.4 Toepassing XPath . . . 49
4.4.1 Standaard XPath . . . 49
4.4.2 Core XPath . . . 51
4.4.3 Minimale XPath expressies en boomautomaten . . . 54
4.4.4 Core XPath en boomautomaten . . . 56
4.4.5 Equivalentie van XPath expressies . . . 57
4.5 Besluit . . . 61

5 Stringautomaten en pattern matching **62**
5.1 Didactische wenken . . . 62
5.2 Wat is pattern matching . . . 62
5.3 Overzicht van algoritmen . . . 63
5.3.1 Naïeve oplossingsmethode . . . 63
5.3.2 Knuth Morris Pratt algoritme (KMP) . . . 64
5.3.3 Het Boyer-Moore algoritme (BM) . . . 66
5.3.4 Zoeken met een deterministische eindige automaat . . . 68
5.3.5 Forward DAWG Matching algorithm (Suffix automaton) . . . 69
5.3.6 Reverse Factor algorithm . . . 71
5.4 Besluit . . . 72

6 Finite state adventures **73**
6.1 Didactische wenken . . . 73
6.2 Wat zijn finite state adventures? . . . 73
6.3 Methode . . . 74
6.3.1 Inleiding . . . 74
6.3.2 Level 0 . . . 75
6.3.3 Level 1 . . . 75
6.3.4 Level 2 . . . 76
6.3.5 Level 3 . . . 77
6.3.6 Level 4 . . . 77
6.4 Besluit . . . 78

7 XViz, een visualisatietool voor XPath **79**
7.1 Didactische wenken 79
7.2 XViz 79
7.2.1 Werkwijze 79
7.2.2 Relaties 81
7.2.3 Het programma 81
7.3 Besluit 82

8 Besluit van deze thesis **83**

Hoofdstuk 1

Inleiding

1.1 Doelstellingen van het vak Theoretische Informatica

De term "theoretische informatica" is de naam die gegeven wordt aan het deelgebied van de informatica dat het meest gemeen heeft met de toegepaste wiskunde. Theoretische informatici zijn zij die bezig zijn met het zoeken van fundamentele oplossingen voor algemene informaticavraagstukken. Het is niet moeilijk om in te zien dat we hier spreken over een zeer uitgebreid onderwerp.

De studenten van het tweede bachelorjaar informatica krijgen een inleiding in de theoretische informatica. Jammer genoeg ontbreekt het veel studenten aan motivatie voor deze cursus. Dit komt waarschijnlijk door het abstracte niveau, de hoge moeilijkheidsgraad en het ontbreken van een duidelijke link met de praktijk. Deze thesis tracht aan de hand van didactische basisprincipes de oorzaak van dit probleem te achterhalen en reikt enkele oplossingen aan.

Via het trajectboek [1] worden aan de leerlingen de doelstellingen van het vak theoretische informatica meegedeeld. Naast praktische doelstellingen zoals toepassingen en implementatie zijn dit de hoofddoelstellingen:

1. De student leert de basisprincipes van reguliere talen
2. De student leert de basisprincipes van context-vrije talen
3. De student leert de basisprincipes van beslisbaarheid en onbeslisbaarheid

Bij reguliere talen krijgen de studenten ook een definitie voor reguliere expressies en de Thompson-constructie. Verder wordt er gesproken over stringtalen en stringautomaten. Onder de toepassingen van context-vrije talen en grammatica's valt ook het CYK parsing algoritme dat de studenten moeten kennen en kunnen toepassen.

1.2 Methodisch-didactische basisprincipes

Er is heel wat geschreven rond didactiek en de methodiek van onderwijzen. In het boek *Psyhodidactiek en het onderwijs* [4] beschrijft Jos Thielemans de belangrijkste methodisch-didactische basisprincipes. Dit zijn vuistregels voor de leerkracht; hij of zij wordt verwacht hier op elk moment in zijn onderwijspraktijk zoveel mogelijk rekening mee te houden.

Een opsomming van deze principes met een korte uitleg:

1. Aanschouwelijkheid
   Om te vermijden dat studenten enkel kunnen praten over de leerstof, zonder deze ooit in werkelijkheid gezien te hebben, is het belangrijk dat de leerinhouden zintuiglijk waarneembaar voorgesteld worden. Het komt er dus op neer zoveel mogelijk aanschouwelijk materiaal te gebruiken in plaats van verbale uitleg.

2. Activiteit
   Leerlingen moeten actief betrokken worden bij het leerproces. De leerkracht fungeert dus als begeleider en stimulator terwijl hij ook moet zorgen voor een gunstig klimaat.

3. Belangstelling
   De leerkracht moet de studenten trachten te motiveren en te interesseren voor de leerstof. Dit kan bijvoorbeeld door voorbeelden te gebruiken uit hun leefwereld en aan te knopen bij hun ervaringen. Ook is het belangrijk om de studenten geregeld (tussentijds) successen te laten beleven om hun belangstelling te behouden.

4. Individualisatie en differentiatie
   Er moet rekening gehouden worden met het feit dat geen twee studenten gelijk zijn en dus ook het leerproces verschilt tussen hen. Dit kan op verschillende niveau's:

   - in doelstellingen: minimale en uitbreidingsdoelen
   - in belangstelling: leerlingen kiezen zelf een deel teksten uit of een onderwerp voor een project
   - in methodische aanpak: afwisseling in didactische werkvormen
   - in tempo: bijvoorbeeld door zelfstandig werk

   Te vaak wordt het hele onderwijsproces afgestemd op de gemiddelde student waardoor zowel zwakkere als sterkere studenten tekortgedaan worden.

5. Integratie
   Kennis moet een geïntegreerd geheel worden en geen samenraapsel van feiten en weetjes. Door verbanden te leggen, verschillende oefeningen te laten maken en te wijzen op de mogelijke toepassingen kan nieuwe kennis geïntegreerd worden met reeds verworven kennis. Deze kennis komt zowel uit dit vak als uit andere vakken.

6. Geleidelijkheid
   Onderwijsdoelstellingen kunnen slechts geleidelijk gerealiseerd worden. Werk van eenvoudige naar complexere doelstellingen en zorg ervoor dat de stap tussen voorkennis en te verwerven kennis niet te groot wordt.

1.3 Indeling van de thesis

Het eerste deel van deze thesis, hoofdstuk 2, omvat de ontwikkeling van een tool die de studenten de kans geeft met behulp van voorbeelden het CYK parsing algoritme te doorgronden. De bedoeling is het algoritme te visualiseren (aanschouwelijk voor te stellen) en enkele oefeningen voor de studenten aan te bieden.

Als uitbreiding op de Thompson-constructie wordt de Glushkov-constructie beschreven in hoofdstuk 3 en wordt er een vergelijking gemaakt tussen deze twee constructies voor automaten van reguliere expressies.

Stringautomaten worden uitgebreid tot boomautomaten in hoofdstuk 4 waarbij de brede toepassingsmogelijkheden (XML, DTD, XPath) worden benadrukt en uitgelegd aan de hand van voorbeelden. Er wordt in dit hoofdstuk ook nog eens teruggegaan naar de eerder geziene Glushkov-constructie en het nut ervan voor de definitie van ondubbelzinnige DTD's.

In het volgende hoofdstuk (5) van deze thesis wordt er een link gelegd tussen de theoretische informatica en het vak algoritmen en datastructuren. Aan de hand van verschillende algoritmen wordt duidelijk hoe stringautomaten kunnen helpen bij pattern matching.

Het voorlaatste hoofdstuk beschrijft een alternatieve toepassing van automaten, formele talen en berekenbaarheid: finite state adventures.

Tot slot wordt in hoofdstuk 7 een korte inleiding gegeven tot XViz, een visualisatietool voor XPath expressies.

Alle hoofdstukken in deze thesis bevatten een korte inleiding waarin didactische wenken opgesomd worden. Deze bevatten persoonlijke opmerkingen over de huidige cursus, tips voor het aanbrengen van de leerstof in de klas en hulpmiddelen voor het motiveren van de studenten.

Hoofdstuk 2

Grammatica's en parsing

2.1 Didactische wenken

Zowel bij de theorie van reguliere talen en automaten (Hoofdstuk 1 van Sipser[3]) als bij grammatica's (Hoofdstuk 2 van Sipser[3]) kan de docent de link leggen tussen de natuurlijke taal en linguïstiek (of programmeertalen en hun syntaxis) enerzijds en de mathematische definities van woord, taal en grammatica anderzijds. Hierdoor maakt de docent een aanknoping bij reeds bestaande kennis, dit past in het *integratieprincipe* en kan zorgen voor een sneller inzicht in de nieuwe structuren. Er moet echter steeds op gelet worden dat de studenten ook de abstracte, mathematische termen begrijpen en zich niet blindstaren op deze voorbeelden of toepassingen. Dit kan door bijvoorbeeld veel verschillende voorbeelden te geven waarbij er geen directe link is met de natuurlijke taal of een programmeertaal zoals een drankautomaat, schuifdeur, avonturen (zie ook hoofdstuk 6), ...

In dit handboek wordt er in het eerste hoofdstuk amper gesproken over de natuurlijke taal of programmeertalen, terwijl in de oude cursus [2] zeer veel voorbeelden uit de natuurlijke taal en programmeertaal werden gebruikt. Het is daarom ook aan te raden een gulden middenweg te zoeken. Zo kan de docent het voorbeeld van programmeertalen als rode draad door de theorie gebruiken waarbij voor elk nieuw begrip (woord, taal, automaat, reguliere expressie, grammatica, afleidingsboom, ...) een voorbeeld gegeven kan worden voor een programmeertaal die de studenten kennen. De uitwerking van deze voorbeelden is terug te vinden in de oude cursus [2].

In hoofdstuk 6 van het trajectboek [1] wordt het CYK parsing algoritme uitgelegd aan de studenten. Hierbij worden jammer genoeg enkele basisprincipes overschreden.

Als eerste kunnen we zeggen dat de *aanschouwelijke* voorstelling van het algoritme ontbreekt. Ondanks het stukje pseudo-code dat uitgeschreven werd, is het niet duidelijk voor de student wat er in elke stap van het algoritme gebeurt, laat staan wat dit betekent voor het parsen. Als oplossing hiervoor gaan we in deel van de thesis een tool implementeren die de studenten meer inzicht moet geven in het CYK parsing algoritme door de aanschouwelijke voorstelling.

Het tweede basisprincipe dat overschreden wordt, is dat van *geleidelijkheid*. Door het verweven van de uitleg over CYK parsing, afleidingsbomen en expressiebomen is het niet meer duidelijk voor de student wat hij aan het leren is. De uitleg over expressiebomen zou volledig

los van de uitleg van het parsing algoritme gegeven moeten worden, in een aparte sectie. Ook moet ervoor gezorgd worden dat de figuren niet te veel door elkaar lopen zodat het overzicht op de cursus behouden blijft. Op die manier kan de student eerst leren wat afleidingsbomen zijn en hoe deze te vinden, om in een volgende stap te gaan kijken naar de expressiebomen.

Als derde ontbreken er opgaven waarbij de student zelf de afleidingsboom van een bepaalde string moet bepalen aan de hand van het algoritme. De studenten zijn bijgevolg niet *actief* bezig met de leerstof. De tool die we gaan implementeren helpt ook hierin aangezien de studenten in kortere tijd meer oefeningen kunnen maken. Ook worden enkele voorbeeldoefeningen aangereikt.

2.2 Contextvrije grammatica

Een contextvrije grammatica [6] of CVG bestaat uit de volgende delen:

- Een verzameling T van terminale symbolen
- Een verzameling N van niet-terminale symbolen; N en T zijn disjunct
- Een verzameling producties van de vorm: $X_0 \rightarrow X_1 \dots X_n$
  waarbij $X_0 \in N$ en $X_i \in N \cup T$ voor $i = 1, ..., n$
- En een startsymbool $S \in N$

Een string s over T voldoet aan de grammatica als we s kunnen afleiden uit het startsymbool S met behulp van de producties. In elke stap van de afleiding vervangen we een niet-terminaal symbool X_0 in de huidige string door de rechterkant $X_1 \dots X_n$ van een productie die X_0 als linkerkant heeft.

2.2.1 Voorbeeld

Beschouw de CVG $G = (N, T, P, S)$ met $N = S, A, B$, $T = a, b$ en P de verzameling van producties:

$$\begin{array}{llll} S \rightarrow \epsilon & S \rightarrow aB & S \rightarrow bA & A \rightarrow aS \\ A \rightarrow bAA & B \rightarrow bS & B \rightarrow aBB & \end{array}$$

Een afleiding voor de string *aabb* waarbij de vetgedrukte variabelen telkens herschreven worden volgens een productie uit G hierboven:

S	$S \rightarrow aB$
a**B**	$B \rightarrow aBB$
aa**B**B	$B \rightarrow bS$
aab**S**B	$S \rightarrow \epsilon$
$aab\epsilon B$	removal of ϵ
aab**B**	$B \rightarrow bS$
$aabb$**S**	$S \rightarrow \epsilon$
$aabb\epsilon$	removal of ϵ
$aabb$	

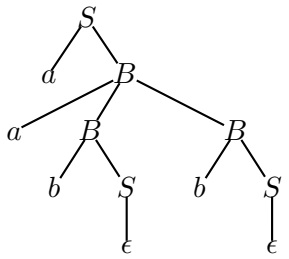
2.3 Parsing

2.3.1 Wat is parsing

Een woord dat voldoet aan een contextvrije grammatica kan grafisch voorgesteld worden met behulp van een parse tree of afleidingsboom. Voor een contextvrije grammatica $G = (V, \Sigma, P, S)$ en een woord $w \in \Sigma^*$ voldoet de afleidingsboom aan:

- de wortel is gelabeld met het startsymbool S
- elke inwendige knoop is gelabeld met een variabele in V
- als een inwendige knoop gelabeld is met A en n kinderen heeft die van links naar rechts gelabeld zijn met $B_1...B_n$, dan moet $A \to B_1...B_n$ een productie zijn in P
- een met ϵ gelabelde knoop is het enige kind van zijn ouder
- de concatenatie van de labels van de bladeren, van links naar rechts, is gelijk aan w.

De afleidingsboom voor de string *aabb* voor de CVG uit 2.2.1 ziet er uit zoals in figuur 2.1.



Figuur 2.1: Afleidingsboom voor aabb

2.3.2 Het CYK parsing algoritme

Het CYK parsing algoritme [1], genoemd naar Cocke, Younger en Kasami, is een algoritme dat voor een gegeven contextvrije grammatica (CVG) in Chomsky normaalvorm (CNF) de afleidingsboom zoekt voor een bepaalde invoer indien deze invoer voldoet aan de grammatica.

Zij $G = (V, \Sigma, P, S)$ een CVG in CNF. Alle producties van P zijn dus van de vorm $A \to BC$ of $A \to a$, met A, B en C in V en a in Σ. Zij w een woord van $\mathcal{L}(G)$ en zij $|w| = n$. Omdat het lege woord niet tot G kan behoren is $n > 0$. Voor $1 \leq i \leq j \leq n$ noteren we nu w_{ij} voor w's deelwoord $w[i]...w[j]$. Verder zeggen we $\forall A$ in V en voor $1 \leq i \leq j \leq n$, dat $D[A, i, j]$ *waar* is als $A \Rightarrow_G^* w_{ij}$ en *onwaar* anders.

Door de vorm van de producties geldt als $i = j$ dat $D[A, i, j] = D[A, i, i]$ *waar* is als en slechts als $A \to w[i]$ tot P behoort. Als $i < j$ daarentegen dan is $D[A, i, j]$ waar als en slechts als er een B en C in V bestaan met $A \to BC$ een productie van P en een getal k met $i \leq k < j$ zodat $D[B, i, k]$ en $D[C, k + 1, j]$ beide *waar* zijn.

Als we de driedimensionale matrix D op bovenstaande manier berekenen, moeten we er steeds voor zorgen dat de componenten ingevuld worden volgens stijgende lengte van de deelwoorden. Met andere woorden, dat $D[B, i, k]$ en $D[C, k + 1, j]$ reeds geëvalueerd zijn voor we $D[A, i, j]$ evalueren. Het woord van lengte n wordt aanvaard als $D[S, 1, n]$ *waar* is.

Om de afleidingsboom te construeren, moeten we nog bijhouden welke producties gebruikt worden. Hiervoor maken we de driedimensionale matrices L,M en R aan. Bij een productie $A \rightarrow BC$ bevat $L[A, i, j]$ het eerste (linkse) symbool in het rechterlid (B), $R[A, i, j]$ bevat het rechtse symbool (C) en $M[A, i, j]$ bevat de positie waar we het woord opsplitsen (k).

De berekening van de matrices D, L, M en R in pseudo-code is te zien in figuur 2.2.

```
for A in V do
  begin
    //initialisatie van de matrices
    for i := 1 to n do
      for j := i to n do
        D[A,i,j] := false;
    //producties van de vorm A -> a
    for i := 1 to n do
      if A -> w[i] in P then
        D[A,i,j] := true
  end;

for j := 2 to n do
  for i := j - 1 downto 1 do
    for A ->BC in P do
      for k := i to j - 1 do
        if D[B,i,k] and D[C,k + 1, j]
          then begin
            D[A,i,j] := true;
            L[A,i,j] := B;
            M[A,i,j] := k;
            R[A,i,j] := C;
          end
```

Figuur 2.2: Pseudo code voor de berekening van D,L,M en R

De uiteindelijke afleidingsboom creëren we door eerst de matrices D, L, M en R te berekenen en vervolgens, als $D[S, 1, n]$ *waar* is, recursief een boom te tekenen. Maak hiervoor eerst een knoop m aan als wortel en voer dan de stappen uit op de volgende pagina met als startwaarden: $A = S, i = 1$ en $j = n$.

1. Label de knoop m met A
2. Als $i = j$:
   - maak een nieuwe knoop n als enige kind van m
   - label de knoop n met $w[i]$
3. Als $i \neq j$
   - maak twee nieuwe knopen n_1 en n_2 als kinderen van m
   - ga terug naar 1. voor $m = n_1, A = L[A, i, j], i = i$ en $j = M[A, i, j]$
   - ga terug naar 1. voor $m = n_2, A = R[A, i, j], i = M[A, i, j] + 1$ en $j = j$

2.3.3 Voorbeeld

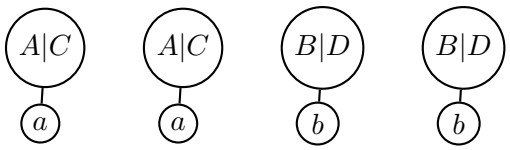
Beschouw het woord *aabb* en de CVG in CNF met volgende producties:

$S \to AB \quad A \to AC \quad A \to a \quad C \to a$
$B \to BD \quad B \to b \quad D \to b$

De eerste for-lus van het algoritme zal de matrices D,L,M en R initialiseren en de unaire producties verwerken. Hierdoor worden alle waarden op $false$ gezet behalve de volgende:

- $D[A, 1, 1]$ en $D[A, 2, 2]$ worden *true* omdat de productie $A \to a$ rechtstreeks toe te passen is op de eerste en tweede letter van het woord *aabb*. Hetzelfde geldt voor $D[C, 1, 1]$ en $D[C, 2, 2]$ met de productie $C \to a$.
- $D[B, 3, 3]$ en $D[B, 4, 4]$ worden *true* omdat de productie $B \to b$ rechtstreeks toe te passen is op de derde en vierde letter van het woord *aabb*. Hetzelfde geldt voor $D[D, 3, 3]$ en $D[D, 4, 4]$ met de productie $D \to b$.

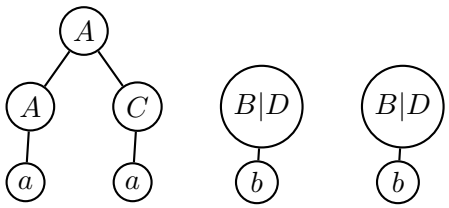
Dit kunnen we grafisch voorstellen zoals in figuur 2.3.



Figuur 2.3: Afleidingsboom voor aabb

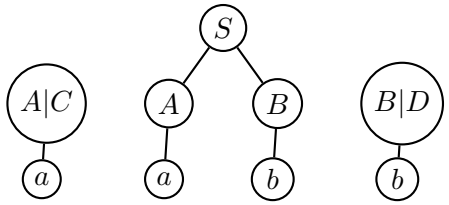
Als we vervolgens het algoritme verder uitvoeren, worden er nog zes wijzigingen doorgevoerd in de matrices:

- $i = 1$, $j = 2$ en $k = 1$ voor de productie $A \to AC$: $D[A, 1, 2]$ wordt *true* met $L = A$, $R = C$ en $M = 1$.
  Dit betekent dat we de productie $A \to AC$ kunnen toepassen op de string *aabb* (van positie 1 tot positie 2 met splitsing na positie 1; dus op a en a), rekening houdend met de reeds verwerkte (unaire) producties. (figuur 2.4)



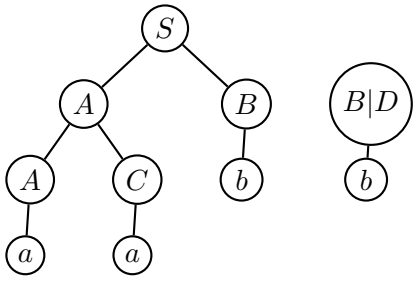
Figuur 2.4: D[A,1,2]

- $i = 2$, $j = 3$ en $k = 2$ voor de productie $S \rightarrow AB$: $D[S, 2, 3]$ wordt *true* met $L = A$, $R = B$ en $M = 2$.
  Dit betekent dat we de productie $S \rightarrow AB$ kunnen toepassen op de string *aabb* (van positie 2 tot positie 3 met splitsing na positie 2; dus op a en b), rekening houdend met de reeds verwerkte (unaire) producties. (figuur 2.5)



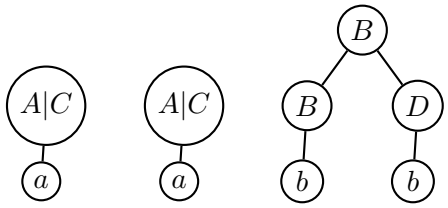
Figuur 2.5: D[S,2,3]

- $i = 1$, $j = 3$ en $k = 2$ voor de productie $S \rightarrow AB$: $D[S, 1, 3]$ wordt *true* met $L = A$, $R = B$ en $M = 2$.
  Dit betekent dat we de productie $S \rightarrow AB$ kunnen toepassen op de string *aabb* (van positie 1 tot positie 3 met splitsing na positie 2; dus op *aa* en b), rekening houdend met de reeds verwerkte producties. Hierbij wordt dus de productie $A \rightarrow AC$ gebruikt op *aa*. Dit weten we doordat $D[A, 1, 2]$ true is met $L = A$, $R = C$ en $M = 1$. (figuur 2.6)



Figuur 2.6: D[S,1,3]

- $i = 3$, $j = 4$ en $k = 3$ voor de productie $B \rightarrow BD$: $D[B, 3, 4]$ wordt *true* met $L = B$, $R = D$ en $M = 3$.
  Dit betekent dat we de productie $B \rightarrow BD$ kunnen toepassen op de string *aabb* (van positie 3 tot positie 4 met splitsing na positie 3; dus op *b* en *b*), rekening houdend met de reeds verwerkte producties. Hierbij worden enkel unaire producties gebruikt. (figuur 2.7)

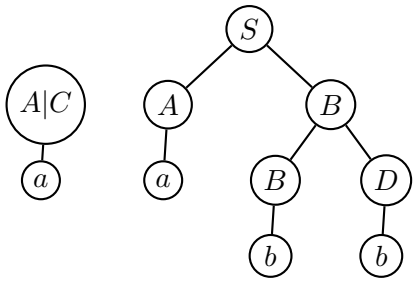


Figuur 2.7: D[B,3,4]

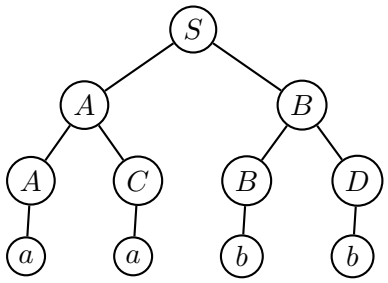
- $i = 2$, $j = 4$ en $k = 2$ voor de productie $S \rightarrow AB$: $D[S, 2, 4]$ wordt *true* met $L = A$, $R = B$ en $M = 2$.
  Dit betekent dat we de productie $S \rightarrow AB$ kunnen toepassen op de string *aabb* (van positie 2 tot positie 4 met splitsing na positie 2; dus op *a* en *bb*), rekening houdend met de reeds verwerkte producties. Hierbij wordt dus de productie $B \rightarrow BD$ gebruikt op *bb*. Dit weten we doordat $D[B, 3, 4]$ true is met $L = B$, $R = D$ en $M = 3$. (figuur 2.8)

- $i = 1$, $j = 4$ en $k = 2$ voor de productie $S \rightarrow AB$: $D[S, 1, 4]$ wordt *true* met $L = A$, $R = B$ en $M = 2$.
  Dit betekent dat we de productie $S \rightarrow AB$ kunnen toepassen op de string *aabb* (van positie 1 tot positie 4 met splitsing na positie 2; dus op *aa* en *bb*), rekening houdend met de reeds verwerkte producties. Hierbij wordt dus de productie $A \rightarrow AC$ gebruikt op *aa*, en $B \rightarrow BD$ gebruikt op *bb*. Dit weten we doordat $D[A, 1, 2]$ en $D[B, 3, 4]$ true zijn. (figuur 2.9)

Na deze stappen is $D[S, 1, 4]$ dus *true* en kunnen we zeggen dat de string *aabb* aanvaard werd. De parse tree ziet er uit zoals in figuur 2.9.



Figuur 2.8: D[S,2,4]



Figuur 2.9: D[S,1,4]

2.4 CYK parsing in JFLAP

2.4.1 Wat is JFLAP?

JFLAP [5] is een pakket met grafische tools die gebruikt kunnen worden als hulp bij het leren van de basisconcepten van formele talen en automatentheorie. Ik werk met de versie van januari 2004, JFLAP4.0b10. In deze versie worden files opgeslagen met behulp van XML. Deze versie gebruikt Java 1.4. In JFLAP kunnen we werken met de volgende concepten:

- Reguliere talen

  Voor het voorstellen van reguliere talen kunnen we eindige deterministische (EDA) en niet-deterministische (ENDA) automaten, reguliere talen en reguliere grammatica's definiëren. We kunnen een ENDA omzetten in een EDA of een minimale EDA. Ook kunnen we de conversie uitvoeren van ENDA naar reguliere expressie of reguliere grammatica en terug. Verder kan de gebruiker de automaten testen met invoer. Dit kan in één keer, per stap of met meervoudige invoer.

- Context vrije talen

  Voor het beschrijven van context vrije talen beschikt JFLAP over twee structuren: de pushdown-automaat en de context vrije grammatica. Ook voor deze structuren heeft JFLAP conversies voorzien. Zo kan je bijvoorbeeld een context vrije grammatica omzetten in Chomsky normaalvorm, in een pushdown-automaat of in een LL of LR parse table. Een pushdown-automaat kan steeds terug omgezet worden in een contextvrije grammatica. JFLAP bevat origineel drie mogelijkheden voor het parsen van een grammatica. De optie Brute Force Parse bekijkt elke mogelijke substitutie van variabelen en duurt daardoor het langst. De twee andere opties, LL(1) en LR(1) parsing, werken met een parse table die gebruikt wordt voor het verwerken van de invoer.

- Turing machines

  Ook voor Turing machines zijn er verscheidene mogelijkheden. Omdat dit buiten het bereik van mijn thesis valt, ga ik hier niet verder op in.

2.4.2 Werkwijze voor CYK parsing in JFLAP

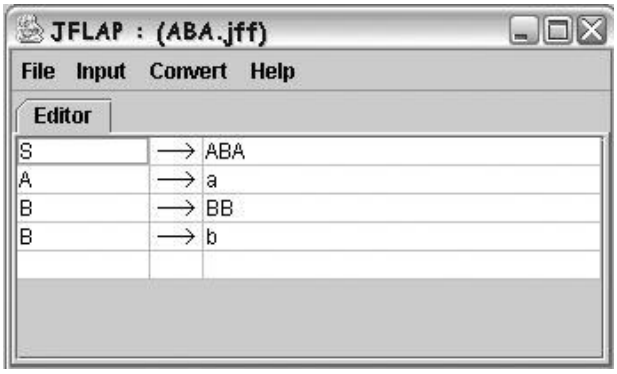
Het invoegen van een grammatica is een van de functionaliteiten van JFLAP. Ik heb een module ingevoegd die het mogelijk maakt een grammatica te parsen met behulp van het CYK parsing algoritme uit paragraaf 2.3.2. In deze sectie wordt de werkwijze uitgelegd.

1. Grammatica

   Als eerste moet de gebruiker de grammatica definiëren waarmee hij wil werken. Hiervoor klikt hij in het startscherm (figuur 2.10) op "Grammar". Vervolgens worden de producties ingevuld: de linkerkant van de productie komt in de eerste kolom, het pijltje volgt automatisch in de tweede kolom, en de rest wordt in de derde kolom ingevuld. Het aantal producties is onbeperkt, er wordt automatisch een nieuwe regel toegevoegd. Als voorbeeld hebben we in JFLAP de volgende grammatica ingevuld: (zie ook in figuur 2.11)

   $S \to ABA \quad A \to a$
   $B \to BB \quad B \to b$

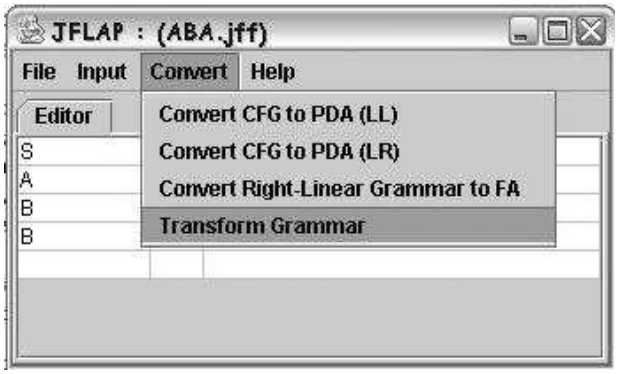
Figuur 2.10: Startscherm van JFLAP



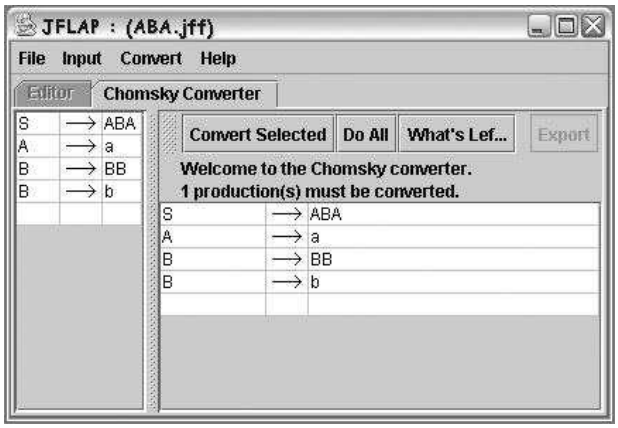
Figuur 2.11: Grammatica invoegen

Als de grammatica is ingevuld, kunnen we hem omzetten in Chomsky normaalvorm door in het menu "Convert"de optie "Transform Grammar"te kiezen (figuur 2.12). Dit is een standaardfunctionaliteit van JFLAP. Voor meer uitleg kan het "Help"-menu geraadpleegd worden. Voor CYK parsing is het voldoende ineens alle stappen uit te voeren ("Do All") en de grammatica te exporteren ("Export") (figuur 2.13). We bekomen dan de volgende grammatica in CNF: (zie ook figuur 2.14)

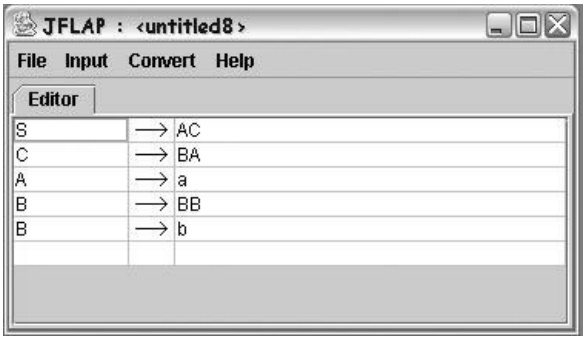
$S \to AC \quad C \to BA \quad A \to a$
$B \to BB \quad B \to b$



Figuur 2.12: Menukeuze voor omzetten in CNF



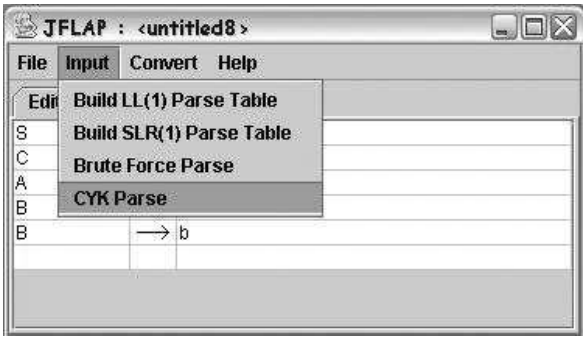
Figuur 2.13: Omzetten in CNF



Figuur 2.14: De grammatica in CNF

3. Scherm

   Vervolgens kiest de gebruiker de optie "CYK Parse" in het menu "Input" (figuur 2.15). Daar krijgt hij een nieuw tabblad te zien waarop de grammatica linksboven zichtbaar blijft. Rechtsboven kan de gebruiker het te parsen woord ingeven in het veld "Input". Beschouw hier als input "abba".

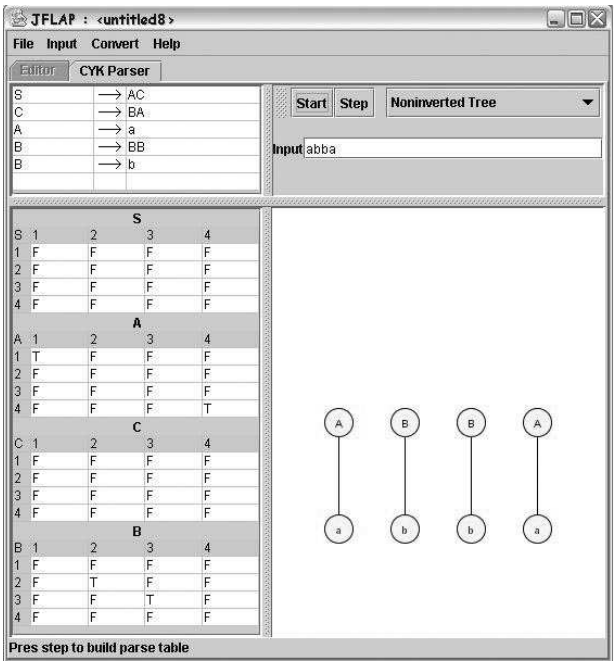


Figuur 2.15: De menukeuze voor CYK parsing

4. Initialisatie van de matrices

   Om het parsen te starten, wordt op de "Start"-knop geklikt of op "return" gedrukt na het invoeren van het woord. Als eerste worden de matrices D, L, M, en R geïnitialiseerd en de unaire producties toegevoegd aan de matrices. In figuur 2.16 zien we linksonder de matrix D. Aangezien het een driedimensionale matrix is, wordt deze opgesplitst in tweedimensionale matrices per niet-terminaal symbool.

   De eerste matrix die getoond wordt is dus $D[S]$ indien S het startsymbool is. De waarden van L, M en R worden later toegevoegd aan de respectieve cellen van D waar $D[A, i, j]$ *true* is. De betekenis van de unaire producties kan rechts afgelezen worden in de grafische voorstelling. Voor elk symbool in het woord wordt aangegeven vanuit welke non-terminals we dit symbool rechtstreeks kunnen bereiken. Deze grafische voorstelling komt overeen met die van het uitgewerkte voorbeeld in sectie 2.3.3.

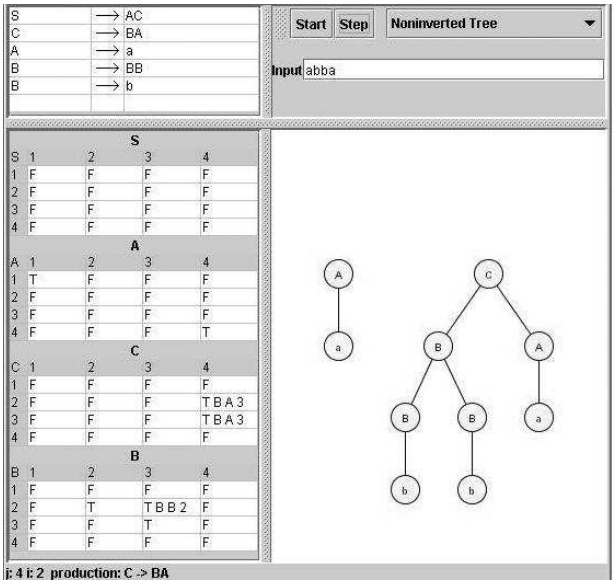


Figuur 2.16: Na de initialisatie

5. Stap voor stap parsen

Na de initialisatie klikt de gebruiker herhaaldelijk op de "Step"-knop, hierbij wordt het parsing algoritme stap voor stap verdergezet. In de statusbalk onderaan kan men de waarde van de variabelen i en j aflezen en kan men zien welke productie net is beschouwd. Indien een stap een wijziging in de matrices tot gevolg heeft, zal dit ook resulteren in een nieuwe grafische voorstelling. Deze voorstelling bestaat, net zoals in sectie 2.3.3, uit een bos waarbij de grootste boom overeenkomt met de net behandelde productie en de kleine bomen de in deze stap onbereikbare symbolen zijn.

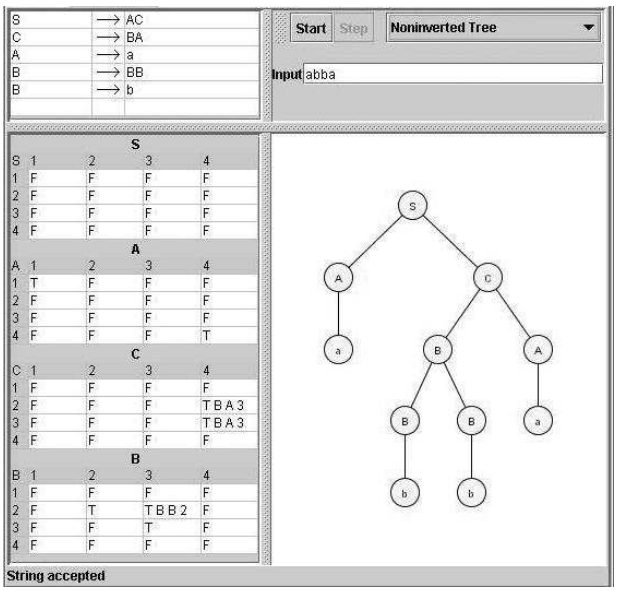
In figuur 2.17 is net de stap uitgevoerd met $j = 4, i = 2$ en als productie $C \rightarrow BA$. Dit houdt in dat men de substring $w[2]..w[4]$ (met $w = abba$ in dit voorbeeld geeft dat de substring bba) kan parsen door te starten bij een knoop C met als kinderen B en A. Als we in de tabel $(C[2, 4])$ gaan kijken zien we dat er gesplitst moet worden na het derde symbool. Dit betekent dan weer dat we vanuit B de substring bb kunnen afleiden en vanuit A het symbool a. Voor knoop A geeft dat een rechtstreekse afleiding, voor knoop B wordt er in de tabel B gekeken op de positie van de substring $([2, 3])$. Hier zien we dat de kinderen B en B zijn, elk met een rechtstreekse afleiding van het symbool b. Deze afleiding toont slechts een substring van het woord, de eerste letter is nog steeds onbereikbaar.



Figuur 2.17: Na een willekeurige stap

6. Resultaat

Als het algoritme kan beslissen over de string, wordt in de statusbalk onderaan aangegeven of de string aanvaard werd of niet. In geval van aanvaarding kan de gebruiker uit de tabellen de afleidingsboom trachten af te leiden. Door nog éénmaal op de "Step"-knop te klikken, wordt de correcte parse tree getoond. In figuur 2.18 ziet u dat de string "abba" aanvaard werd door de grammatica en ziet u de afleidingsboom.



Figuur 2.18: Het eindresultaat

2.5 Oefeningen

Aangezien in de cursus geen enkele oefening werd opgenomen voor het CYK parsing algoritme worden er hier een paar toegevoegd:

2.5.1 Oefening 1

Beschouw de volgende grammatica:

$S \rightarrow \epsilon \quad S \rightarrow aB \quad S \rightarrow bA \quad A \rightarrow aS$
$A \rightarrow bAA \quad B \rightarrow bS \quad B \rightarrow aBB$

Zet deze grammatica om in CNF indien nodig en parse de volgende strings met behulp van JFLAP:

$\epsilon \quad aabb \quad ab$
$a \quad aab \quad abb$

Geef de afleidingsboom indien deze bestaat, welke woorden worden aanvaard?

2.5.2 Oefening 2

Beschouw de volgende grammatica:

$S \rightarrow AB \quad S \rightarrow BC \quad A \rightarrow AA \quad A \rightarrow a$
$B \rightarrow BB \quad B \rightarrow b \quad C \rightarrow CC \quad C \rightarrow c$

Zet deze grammatica om in CNF indien nodig en parse de volgende strings met behulp van JFLAP:

$\epsilon \quad aabb \quad bcc$
$ac \quad abc \quad ccc$

Geef de afleidingsboom indien deze bestaat, welke woorden worden aanvaard?

2.6 Besluit

Met de geïmplementeerde uitbreiding voor JFLAP kunnen studenten stap per stap het CYK parsing algoritme doorlopen zonder daarbij alles op papier te moeten bijhouden. Hierdoor kunnen ze op korte tijd verschillende voorbeelden of herhaaldelijk hetzelfde voorbeeld uitwerken waardoor ze zelf een beeld kunnen vormen van het algoritme. Ook is het mogelijk dat de student zelf de parse-tree heeft berekend zonder tool, en met behulp van de tool een zelfcontrole uitvoert.

Het programma levert de student aan de ene kant een tijdswinst op en aan de andere kant is hij of zij actief betrokken bij het leergebeuren. Het algoritme wordt beter onthouden als de student zelf achter de werkwijze kan komen. De visuele voorstelling van elke stap tijdens het parsen (tussentijdse parse trees) zorgt ook voor meer diepgang.

Hoofdstuk 3

Stringautomaten en reguliere expressies

3.1 Didactische wenken

In het handboek voor het vak theoretische informatica [3] wordt in het eerste hoofdstuk aan de lezer uitgelegd hoe hij een automaat kan maken van een reguliere expressie. Dit algoritme wordt niet benoemd en komt voor in een bewijs. Studenten die dit bewijs bestuderen weten bijgevolg niet dat het hier gaat over de Thompson-constructie. Indien later naar deze constructie verwezen wordt hebben de studenten bijgevolg geen aanknoping. Verder beschouwen veel studenten bewijzen als de moeilijkste elementen uit een cursus. Ze zullen geneigd zijn het bewijs, en bijgevolg ook de constructie, slechts oppervlakkig te bekijken en verliezen hun belangstelling.

Positief is de aanwezigheid van voldoende voorbeelden en hun grafische voorstelling (aanschouwelijkheid), alsook de opgave in het trajectboek voor zelf automaten te construeren van enkele reguliere expressies (activiteit). Ook het opsplitsen van het bewijs in twee richtingen maakt het voor de student eenvoudiger te begrijpen en steunt op het principe van geleidelijkheid.

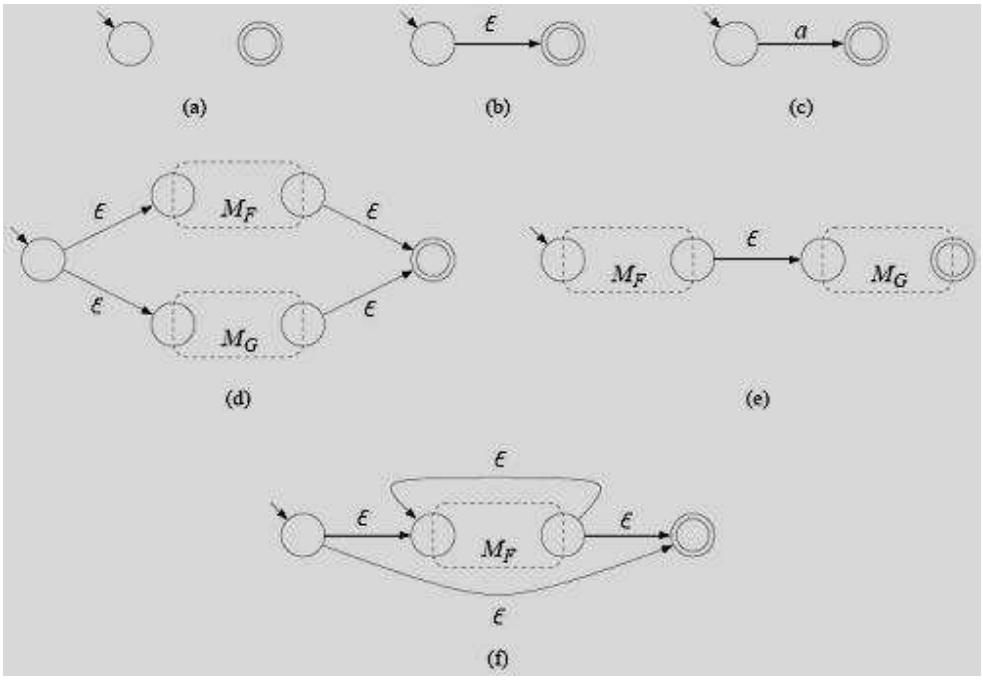
De rest van dit hoofdstuk geeft een vergelijking tussen twee verschillende automaten van reguliere expressies. De Thompson-automaat, die we hierboven reeds vermeld hebben, en de Glushkov-automaat. Deze theorie wordt toegepast in sectie 4.3.5 op pagina 46. Alles wordt uitgelegd aan de hand van eenzelfde voorbeeld zodat de verschillen tussen de twee automaten duidelijk zichtbaar zijn.

3.2 Thompson-constructie

In de definitie van reguliere expressies (zie referenties [2, 3]) wordt niet expliciet vermeld waarom deze expressies juist regulier heten. We weten wel dat een taal regulier heet als er een automaat bestaat die deze taal aanvaardt. Wat is dan het verband tussen reguliere talen en regulieren expressies? Zoals we al kunnen vermoeden is een taal alleen regulier als er een reguliere expressie is die deze taal beschrijft. Ook is de taal die een reguliere expressie beschrijft steeds een reguliere taal. We kunnen dus van elke reguliere expressie een eindige deterministische automaat maken en van elke eindige automaat een reguliere expressie. Dit doen we met behulp van de Thompson-constructie [7, 9].

Het basisidee van de Thompson-constructie is het opsplitsen van de reguliere expressie in minimale deelexpressies die gecombineerd worden met behulp van de unie, concatenatie en Kleene ster. In figuur 3.1 ziet u deze *inductieve* constructie die als resultaat een eindige automaat geeft met de volgende eigenschappen:

- Er is slechts één begintoestand waaruit enkel pijlen vertrekken
- Er is slechts één eindtoestand waarin enkel pijlen toekomen
- Elke toestand heeft maximaal twee inkomende en twee uitgaande pijlen
- De automaat is ten hoogste drie keer zo groot als de reguliere expressie
- De automaat zit vol ϵ-transities.

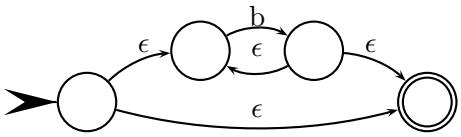


Figuur 3.1: De Thompson-constructie voor een reguliere expressie E: (a.) $E = \emptyset$ (b.) $E = \epsilon$ (c.) $E = a, a \in \Sigma$ (d.) $E = F + G$ (e.) $E = F \cdot G$ en (f.) $E = F^*$

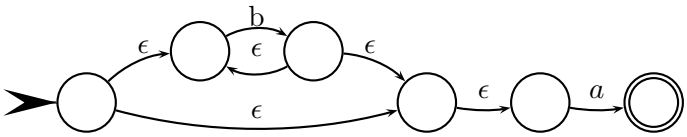
3.2.1 Voorbeeld: Thompson-automaat voor $(b^*a(b^*a)^*) + \epsilon$

We bouwen de automaat voor $b^*a(b^*a)^* + \epsilon$ op als volgt:

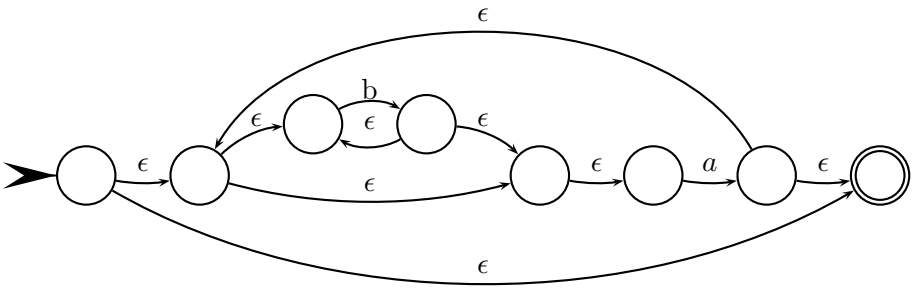
1. b^* zie figuur 3.2
2. b^*a zie figuur 3.3
3. $(b^*a)^*$ zie figuur 3.4
4. $b^*a(b^*a)^*$ zie figuur 3.5
5. $(b^*a(b^*a)^*) + \epsilon$ zie figuur 3.6



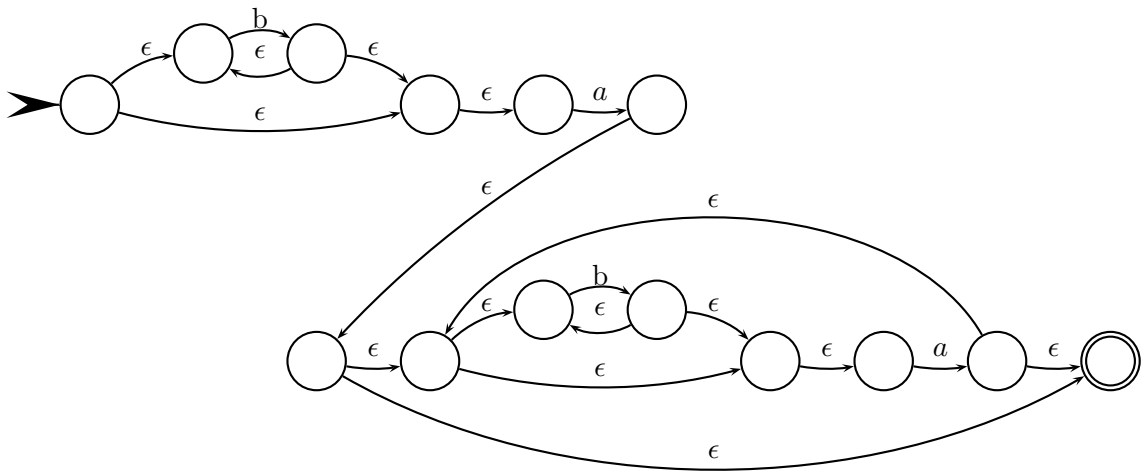
Figuur 3.2: Thompson-automaat voor b^*



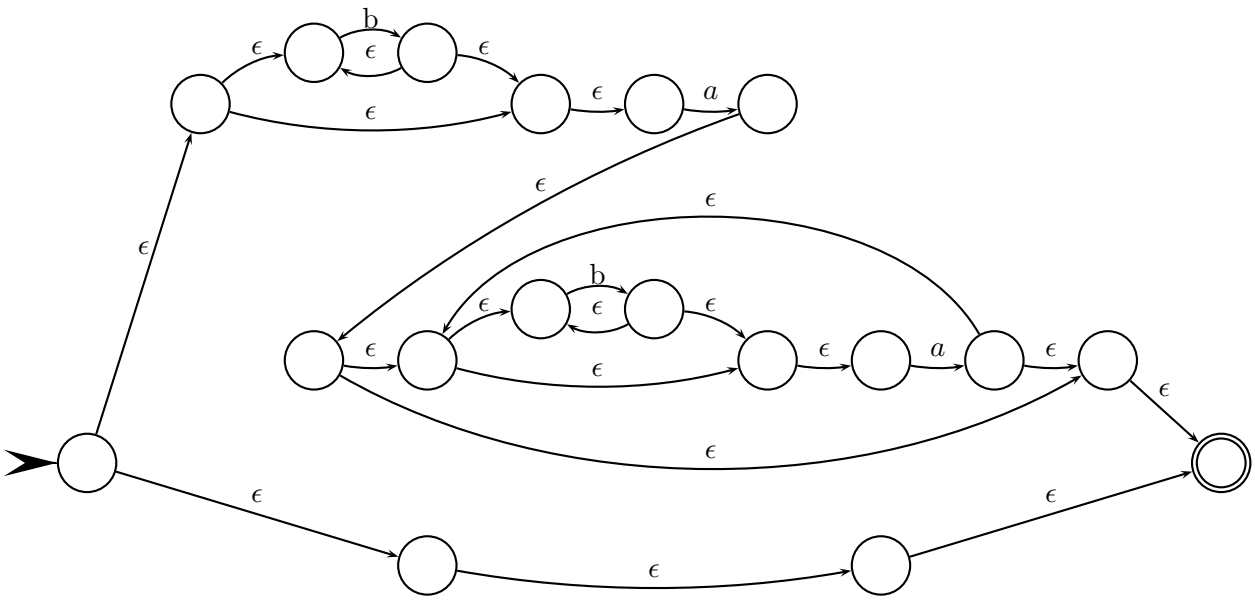
Figuur 3.3: Thompson-automaat voor b^*a



Figuur 3.4: Thompson-automaat voor $(b^*a)^*$



Figuur 3.5: Thompson-automaat voor $b^*a(b^*a)^*$



Figuur 3.6: Thompson-automaat voor $(b^*a(b^*a)^*) + \epsilon$

3.3 Glushkov-constructie

In [8] bespreken Brüggemann–Klein en Wood een andere methode om een reguliere expressie om te zetten in een eindige automaat, de zogenaamde Glushkov-constructie [10].

Beschouw volgende definities voor elke taal L:

- $first(L) = \{b \mid \text{er is een woord } w \text{ zodanig dat } bw \in L\}$
- $last(L) = \{b \mid \text{er is een woord } w \text{ zodanig dat } wb \in L\}$
- $follow(L, a) = \{b \mid \text{er zijn woorden } v, w \text{ zodanig dat } vabw \in L\}$ voor elk symbool a
- voor elke expressie E breiden we deze sets uit tot: $first(E) = first(L(E))$ enz.

en voor elke expressie E:

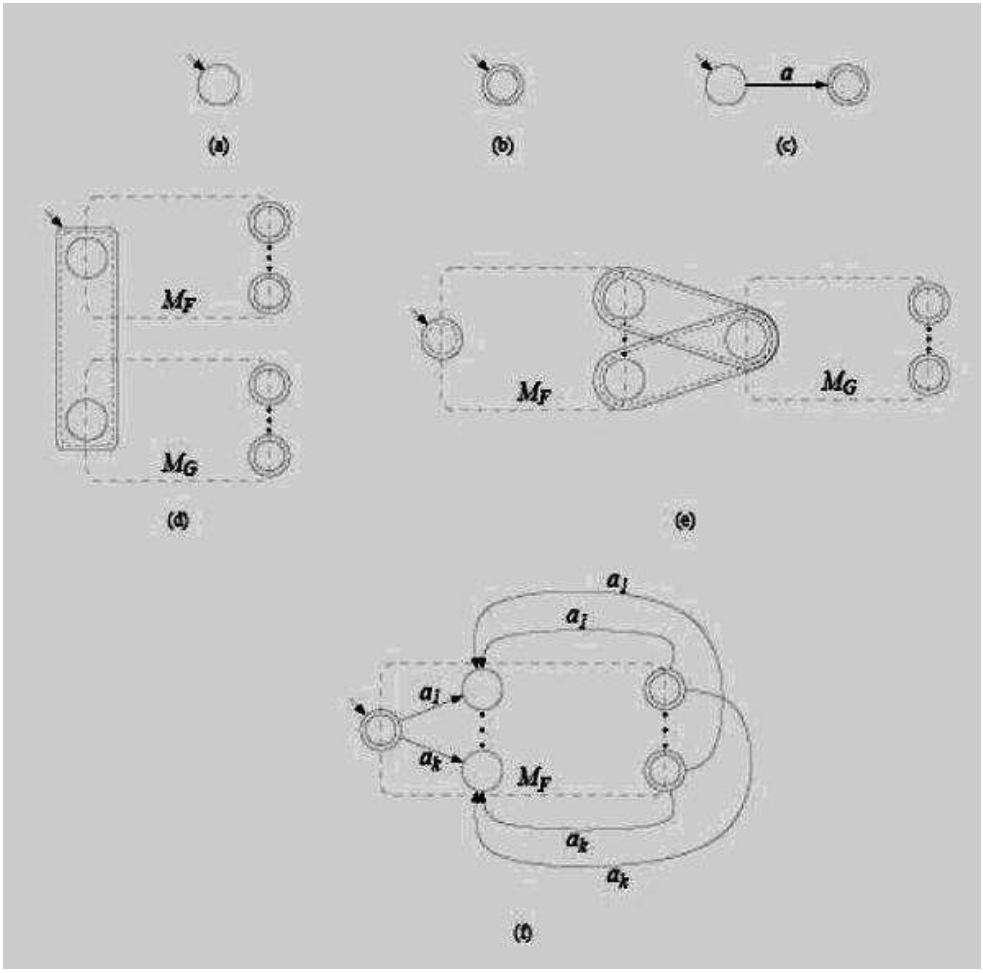
- $sym(E)$ is de verzameling van alle symbolen in de expressie E
- E' is een markering van E waarbij elk symbool een subscript meekrijgt zodanig dat alle symbolen van E' slechts éénmaal voorkomen
- Π is de verzameling van alle symbolen met subscript $\supset sym(E')$
- E^t is de notatie voor de expressie zonder subscripts met ook a^t het symbool zonder zijn subscript.

Bijvoorbeeld voor de expressie $E = (a+b)^*ab$ wordt $sym(E) = \{a, b\}$, $E' = (a_1 + b_1)^*a_2b_2$ en Π bevat onder andere a_1, a_2, b_1 en b_2. Ook kunnen we zeggen dat $a_1^t = a_2^t = a$.

We definiëren de *Glushkov-automaat* $G_E = (Q_E, \Sigma, \delta_E, q_I, F_E)$ van een expressie E dan als volgt:

- $Q_E = sym(E') \cup \{q_I\}$;
  De toestanden zijn de symbolen van de expressie met subscripts, uitgebreid met een nieuwe begintoestand q_I.
- $\delta_E(q_I, a) = \{x \mid x \in first(E'), x^t = a\} \forall a \in \Sigma$
  Vanuit de begintoestand vertrekken er pijlen naar alle toestanden die overeenkomen met een symbool dat als eerste gelezen mag worden. Dit symbool, zonder subscript, wordt als label aan de pijl toegekend.
- $\delta_E(x, a) = \{y \mid y \in follow(E', x), y^t = a\} \forall x \in sym(E')$ en $a \in \Sigma$
  Voor alle toestanden x (met subscript) en alle symbolen a (zonder subscript) vertrekt er een pijl met label a vanuit toestand x naar alle toestanden y die overeenkomen met een symbool dat kan volgen op x in E' en waarbij $y^t = a$.
- $F_E = last(E') \cup \{q_I\}$ als $\epsilon \in L(E)$ en $F_E = last(E')$ anders
  De eindtoestanden komen overeen met de toestanden van alle symbolen waar de expressie op kan eindigen, uitgebreid met q_I als de lege string ook aanvaard wordt.

Om de Glushkov-constructie straks te kunnen vergelijken met de Thompson-constructie van 3.2 gaan we deze op gelijkaardige wijze, inductief, definiëren. Deze constructie van de Glushkov-automaat wordt de Champarnaud-constructie genoemd en is terug te vinden in figuur 3.7 uit [7, 9].



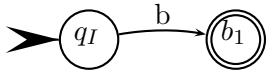
Figuur 3.7: De inductieve constructie van Chapmparnaud van de Glushkov-machine voor een reguliere expressie E: (a.) $E = \emptyset$ (b.) $E = \epsilon$ (c.) $E = a, a \in \Sigma$ (d.) $E = F + G$ (e.) $E = F \cdot G$ en (f.) $E = F^*$

3.3.1 Voorbeeld: Glushkov-automaat voor $(b^*a(b^*a)^*) + \epsilon$

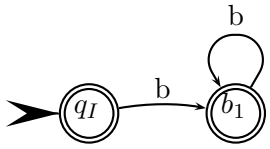
We bouwen de automaat voor $(b^*a(b^*a)^*) + \epsilon$ inductief op. Deze constructie vraagt eigenlijk niet dat we de expressie markeren zoals dat wel gevraagd wordt bij de niet-inductieve constructie. Toch gaan we ook hier eerst de expressie markeren. Hierdoor vermijden we dat we bij het samenvoegen van de deelautomaten toestanden moeten hernoemen. Een automaat kan namelijk geen twee toestanden met dezelfde naam hebben.

We bouwen dus de Glushkov-automaat voor de gemarkeerde expressie $E' = (b_1^* a_1 (b_2^* a_2)^*) + \epsilon$:

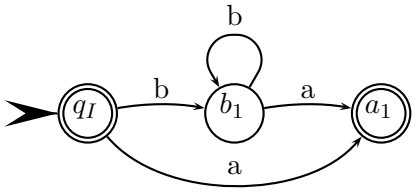
1. b_1 zie figuur 3.8, b_2 is analoog
2. b_1^* zie figuur 3.9, b_2^* is analoog
3. $b_1^* a_1$ zie figuur 3.10, $b_2^* a_2$ analoog
4. $(b_2^* a_2)^*$ zie figuur 3.11
5. $b_1^* a_1 (b_2^* a_2)^*$ zie figuur 3.12
6. $(b_1^* a_1 (b_2^* a_2)^*) + \epsilon$ zie figuur 3.13



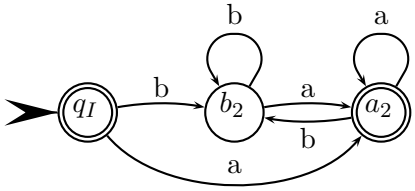
Figuur 3.8: Glushkov-automaat voor b_1



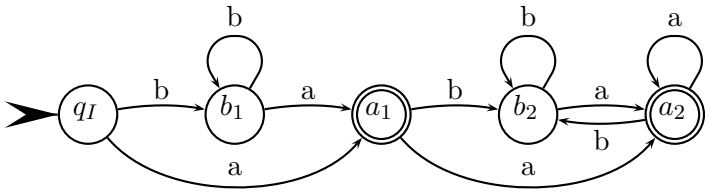
Figuur 3.9: Glushkov-automaat voor b_1^*



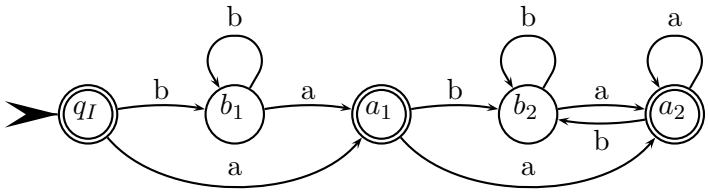
Figuur 3.10: Glushkov-automaat voor $b_1^* a_1$



Figuur 3.11: Glushkov-automaat voor $(b_2^* a_2)^*$



Figuur 3.12: Glushkov-automaat voor $b_1^* a_1 (b_2^* a_2)^*$



Figuur 3.13: Glushkov-automaat voor $(b_1^* a_1 (b_2^* a_2)^*) + \epsilon$

3.3.2 Expliciete constructie van Glushkov-automaat voor $E = (b^* a (b^* a)^*) + \epsilon$

Voor de duidelijkheid berekenen we de automaat voor onze voorbeeldexpressie ook op niet-inductieve wijze. Dit is zoals de Glushkov-automaat eerst gedefinieerd werd in dit hoofdstuk met behulp van $first$, $last$ en $follow$.

- $E' = (b_1^* a_1 (b_2^* a_2)^*)$
- $First(E') = \{a_1, b_1\}$
- $Last(E') = \{a_1, a_2\}$
- $Follow(E', a_1) = \{a_2, b_2\}$
- $Follow(E', a_2) = \{a_2, b_2\}$
- $Follow(E', b_1) = \{a_1, b_1\}$
- $Follow(E', b_2) = \{a_2, b_2\}$

Glushkov-automaat $G = (Q, \Sigma, \delta, q_I, F)$ met:

- $Q = sym(E') \cup \{q_I\} = \{q_I, a_1, a_2, b_1, b_2\}$
- $\delta(q_I, a) = \{x \mid x \in first(E'), x^t = a\} \forall a \in \Sigma$
  - $\delta(q_I, a) = \{a_1\}$
  - $\delta(q_I, b) = \{b_1\}$
- $\delta(x, a) = \{y \mid y \in follow(E', x), y^t = a\} \forall x \in sym(E')$ en $a \in \Sigma$
  - $\delta(a_1, a) = \{a_2\}$
  - $\delta(a_1, b) = \{b_2\}$
  - $\delta(b_1, a) = \{a_1\}$
  - $\delta(b_1, b) = \{b_1\}$
  - $\delta(a_2, a) = \{a_2\}$
  - $\delta(a_2, b) = \{b_2\}$
  - $\delta(b_2, a) = \{a_2\}$
  - $\delta(b_2, b) = \{b_2\}$
- $F = last(E') \cup \{q_I\}$ als $\epsilon \in L(E)$ en $F = last(E')$ anders
  $F = \{q_I, a_1, a_2\}$

Aangezien dit een vrij eenvoudige reguliere expressie is, kunnen we de elementen van $first$, $last$ en $follow$ door observatie uit de expressie halen. Als de expressie langer en/of meer ingewikkeld is, zal dit niet meer zo voor de hand liggen. In [9] wordt er een methode aangereikt om deze verzamelingen in polynomiale tijd te berekenen, inductief op de lengte van de expressie. Daarbij wordt gebruik gemaakt van de parse tree voor de expressie (zie 2.3). We gaan hier niet verder in op deze constructie aangezien we al een methode besproken hebben om de Glushkov-automaat inductief te berekenen waarbij deze verzamelingen niet meer nodig zijn.

3.3.3 Eigenschappen van de Glushkov-automaat

Als we de Glushkov-automaat en zijn constructie bekijken kunnen we volgende eigenschappen noteren:

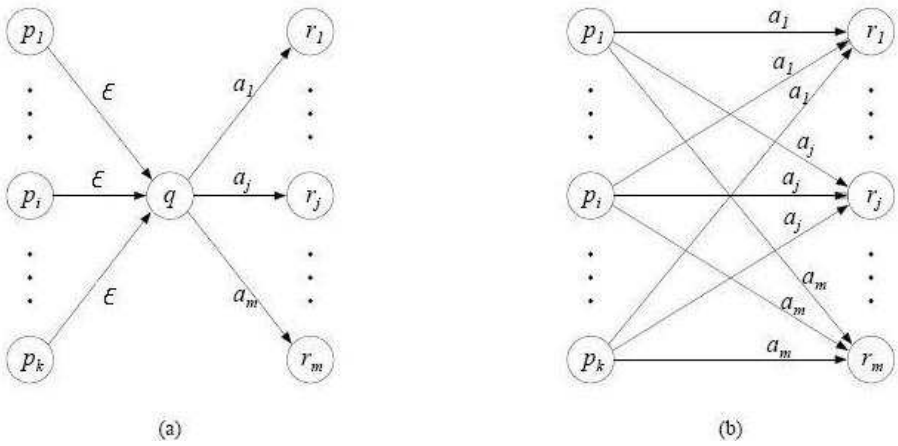
- Er is slechts één begintoestand waaruit enkel pijlen vertrekken
- Er zijn geen stille overgangen
- Voor elke toestand q hebben de inkomende pijlen steeds hetzelfde label
- De grootte van de Glushkov-machine is in het slechtste geval kwadratisch ten opzichte van de lengte van de reguliere expressie
- De Glushkov-automaat bevat geen ϵ-transities

3.4 Vergelijking

In de vorige secties beschreven we twee verschillende algoritmen om van een gegeven reguliere expressie een eindige automaat te maken. Alhoewel de resulterende automaten op het eerste zicht heel verschillend zijn, kunnen we van elke Thompson-automaat de overeenkomstige Glushkov-automaat afleiden. Ook kunnen we de Thompson-automaat vinden met behulp van de Glushkov-afleiding. Merk daarvoor in de eerste plaats op dat zowel de Thompson als de Glushkov-automaat homogene automaten zijn, dat wil zeggen dat in elke toestand enkel pijlen met een en hetzelfde label kunnen binnenkomen. Het verband tussen deze twee constructies wordt ook uitgelegd in de literatuur [7, 9].

3.4.1 Van Thompson naar Glushkov

Als we de Thompson-automaat bestuderen, bemerken we de vele "stille overgangen" of ϵ-transities. In de Glushkov-constructie daarentegen vinden we geen enkele stille overgang. Daarom zoeken we een methode om deze stille overgangen te verwijderen. Aangezien een Thompson-automaat homogeen is zullen de knopen waarin ϵ-transities aankomen (ϵ-knoop) ook verwijderd mogen worden (er komt namelijk geen enkele pijl meer in aan, deze toestanden zijn dus onbereikbaar). Het verwijderen van deze knopen en pijlen komt er op neer elke binnenkomende pijl van een ϵ-knoop rechtstreeks te verbinden met alle uitgaande pijlen, zoals ook zichtbaar is in figuur 3.14. Let wel op wanneer de verwijderde ϵ-knoop een eindtoestand was, dan worden alle toestanden van waaruit we deze knoop konden bereiken, ook eindtoestanden.



Figuur 3.14: De eliminatie van ϵ transities en -knopen

Wanneer we deze eliminatie toepassen op een Thompson-automaat zodat er geen stille overgangen meer voorkomen, merken we dat we exact de Glushkov-automaat bekomen (op de benaming van toestanden na). Dit wordt bewezen in [7].

Voorbeeld

Passen we deze eliminatie toe op de Thompson-automaat van 3.2.1 op pagina 24, bekomen we de Glushkov-automaat van 3.3.1 op pagina 27 mits hernoemen van de toestanden.

3.4.2 Van Glushkov naar Thompson

Kunnen we dan ook een manier vinden om van de Glushkov-automaat naar de Thompson-automaat te gaan? In tegenstelling tot de vorige constructie moeten we hierbij informatie (pijlen en toestanden) toevoegen aan de automaat in plaats van verwijderen. Dit is echter niet zo evident maar in [7] hebben ze een oplossing gevonden. In plaats van de Glushkov-automaat van de reguliere expressie uit te breiden, gaan ze eerst de reguliere expressie zelf uitbreiden om vervolgens de Glushkov-automaat van deze uitgebreide expressie te construeren.

Zij τ een nieuw symbool dat nog niet in het alfabet Σ zit, de *τ-uitbreiding* van een reguliere expressie E over Σ is dan een reguliere expressie E_τ over $\Sigma \cup \{\tau\}$ met (inductief):

- $E_\tau = \emptyset$ if $E = \emptyset$
- $E_\tau = \tau$ if $E = \epsilon$
- $E_\tau = a$ if $E = a$
- $E_\tau = ((\tau \cdot F_\tau) + (\tau \cdot G_\tau)) \cdot \tau$ if $E = F + G$
- $E_\tau = (F_\tau \cdot \tau) \cdot G_\tau$ if $E = F \cdot G$
- $E_\tau = ((\tau \cdot F_\tau)^*) \cdot \tau$ if $E = F^*$
- $E_\tau = (F_\tau)$ if $E = (F)$

Indien we deze uitbreiding toepassen op de reguliere expressie en vervolgens de τ vervangen door ϵ zal de resulterende Glushkov-automaat exact gelijk zijn aan de Thompson-automaat voor de originele reguliere expressie.

Voorbeeld

Beschouw de reguliere expressie $(b^*a(b^*a)^*) + \epsilon$ die we ook gebruikten als voorbeeld voor de Thompson (3.2.1) en Glushkov (3.3.1) constructies. Uit deze expressie kunnen we de τ-uitbreiding als volgt afleiden:

$$
\begin{array}{cl}
(b^* \cdot a \cdot (b^* \cdot a)^*) + \epsilon & \\
\Updownarrow & E = \epsilon \text{ en } E = a \text{ en } E = b \\
(b^* \cdot a \cdot (b^* \cdot a)^*) + \tau & \\
\Updownarrow & E = F + G \\
(\tau(b^* \cdot a \cdot (b^* \cdot a)^*) + \tau\tau)\tau & \\
\Updownarrow & E = F \cdot G \text{ twee maal met } F = b^* \text{ en } G = a \\
(\tau((b^*)\tau a \cdot ((b^*)\tau a)^*) + \tau\tau)\tau & \\
\Updownarrow & E = F^* \text{ twee maal met } F = b \\
(\tau(((\tau b)^*\tau)\tau a \cdot (((\tau b)^*\tau)\tau a)^*) + \tau\tau)\tau & \\
\Updownarrow & E = F^* \text{ met } F = (((\tau b)^*\tau)\tau a) \\
(\tau(((\tau b)^*\tau)\tau a \cdot (\tau((\tau b)^*\tau)\tau a)^*\tau) + \tau\tau)\tau & \\
\Updownarrow & E = F \cdot G \\
(\tau((((\tau b)^*\tau)\tau a)\tau(\tau((\tau b)^*\tau)\tau a)^*\tau) + \tau\tau)\tau & \\
\Updownarrow & \text{overtollige haakjes verwijderen} \\
(\tau((\tau b)^*\tau\tau a\tau(\tau(\tau b)^*\tau\tau a)^*\tau) + \tau\tau)\tau &
\end{array}
$$

Waarbij we voor de duidelijkheid $F \cdot G$ schrijven als FG van het moment we de uitbreiding hebben toegepast op het product. Passen we vervolgens op deze uitgebreide reguliere expressie de Glushkov-constructie toe, dan krijgen we exact de Thompson-automaat voor de originele reguliere expressie $(b^*a(b^*a)^*) + \epsilon$. Deze wordt getoond in figuur 3.6 op pagina 25.

3.4.3 Gebruik van de twee constructies

Kenmerkend voor Thompson-automaten is de aanwezigheid van ϵ-transities. Een Thompson-automaat is een eindige niet-deterministische automaat met ϵ-transities, of een $ENDA - \epsilon$. Deze automaten zijn niet eenvoudig te implementeren aangezien het bereiken van de eindtoestand afhangt van het volgen van de juiste ϵ-transities. Deze constructie is echter zeer geschikt voor didactisch gebruik aangezien het een eenvoudige constructie is die grafisch zeer duidelijk voor te stellen is.

Glushkov-automaten daarentegen bevatten geen ϵ-transities en zijn daardoor beter geschikt voor implementatie. Ze zijn echter minder geschikt als didactisch voorbeeld omwille van de meer wiskundige benadering van de constructie.

3.5 Besluit

De Thompson en Glushkov-constructies zijn twee algoritmen om een reguliere expressie om te zetten in een automaat. Ondanks de mogelijkheid via de ene constructie de andere automaat te bekomen hebben ze niet dezelfde automaat als resultaat. Zo zien we bijvoorbeeld dat de Thompson-automaat altijd groter of gelijk is aan de Glushkov-automaat, aangezien we enkel toestanden elimineren en niet bijmaken om de Glushkov-automaat te bereiken.

Anderzijds is de Thompson-automaat beperkt tot driemaal de grootte van de reguliere expressie en de Glushkov tot het kwadraat van de grootte. Hierdoor kunnen we afleiden dat, voor expressies met lengte groter als drie, de Glushkov-automaat ook steeds kleiner is als drie maal de lengte van de reguliere expressie.

Om bepaalde eigenschappen van de reguliere expressie te achterhalen, is het soms nodig de Glushkov-automaat te construeren. Zo bepaalt het al dan niet deterministisch zijn van de Glushkov-automaat of een reguliere expressie al dan niet one-unambiguous is (zie ook sectie 4.3.5). Met de resultaten in dit hoofdstuk hebben we een alternatieve manier gevonden voor de berekening van de Glushkov-automaat. Indien de expressie unambiguous is, kunnen we de Glushkov-automaat zelfs in lineaire tijd berekenen.

Hoofdstuk 4

Boomautomaten

4.1 Didactische wenken

4.1.1 Differentiatie

De technologie staat niet stil. Er worden constant nieuwe technieken ontwikkeld en verbeteringen doorgevoerd in bestaande technologiën. Bij het schrijven van deze thesis zijn sommige onderwerpen misschien zelfs al verouderd. De stringautomaten die gisteren aangeleerd werden zijn vandaag misschien nog nuttig maar worden morgen weer vervangen door boomautomaten. Boomautomaten blijken momenteel al van groot nut voor het modelleren van relatief nieuwe tekstformaten zoals XML en zullen later zeker nog meer toepassingen krijgen.

In het informaticaonderwijs staan docenten voor de opdracht deze nieuwe ontwikkelingen aan te leren aan de studenten zonder reeds bestaande cursussen en onderwerpen te schrappen. Oude technieken zijn namelijk een goede basis om op verder te bouwen en zullen ook niet zomaar verdwijnen. Maar hierdoor gaan cursussen natuurlijk blijven groeien en verhoogt de studietijd per vak.

Dit geeft de docent echter een ideale kans om aan differentiatie te doen. Alle studenten krijgen dezelfde basis maar nadat deze basis gezien is, kan de docent de keuze aan de student laten. Dit kan enerzijds binnen éń vak in één jaar gebeuren door de student als opdracht een aantal topics te laten bestuderen uit een lijst opgesteld door de docent (keuzeonderwerp). Anderzijds kunnen uitbreidingen gegeven worden in een keuzevak het jaar of semester na het basisvak, waardoor studenten enkel dieper ingaan op onderwerpen in het door hun gekozen domein. Natuurlijk zijn ook projecten en groepswerken ideale methoden om studenten aan verschillende onderwerpen te laten werken.

4.1.2 Integratie

De theorie van boomautomaten steunt in de eerste plaats op het vak Theoretische Informatica. Maar als we naar de mogelijke toepassingen ervan kijken, komen we in het vaarwater van Technologie van Multimediasystemen en -software [12]. Dit onderwerp biedt docenten van beide vakken de mogelijkheid de kennis van hun studenten beter te integreren. Studenten zijn nogal snel geneigd om in vakjes te blijven denken en zullen niet snel verbanden en mogelijke toepassingen zien tussen kennis uit twee verschillende vakken.

Voor intergratie tussen twee verschillende vakken is het wel zeer belangrijk dat docenten overleg plegen. Hierdoor kunnen inconsistenties vermeden worden. Ook moet er duidelijk gesteld worden welke voorkennis uit welke vakken gehaald kan worden, en wanneer de vakken gegeven worden. Indien er bijvoorbeeld gesteund wordt op kennis uit een vak van twee jaar ervoor, is het misschien aan te raden deze eerst op te frissen vooraleer er verder op te bouwen.

4.2 Boomautomaten

4.2.1 Boomautomaat en stringautomaat

Een boomautomaat (tree automaton) is een uitbreiding van een eindige acceptor (finite state automaton). Een eindige acceptor bereikt een nieuwe toestand vanuit één staat met het lezen van één term (karakter) en beslist over de acceptatie van een woord of string. Een boomautomaat bereikt echter een nieuwe toestand vanuit meerdere toestanden en leest meerdere termen tegelijk in. Een boomautomaat beslist over de acceptatie van een boom. Net zoals bij string automaten bestaan er deterministische en niet-deterministische boomautomaten. Een nieuw verschijnsel is het bestaan van bottom-up en top-down automaten. Het verschil zit in het doorlopen van de boom; een bottom-up automaat leest een boom vanuit de bladeren naar de wortel terwijl een top-down automaat zal vertrekken van de wortel. Verder bestaan er ook two-way boomautomaten waarbij er zowel omhoog (ascending of bottom-up) als omlaag (descending of top-down) gewerkt kan worden.[18].

Zoals in [12] beschreven, kunnen we een XML bestand voorstellen als een boom. Deze bomen zijn geen binaire bomen aangezien elke knoop meerdere kinderen kan hebben, het aantal kinderen per knoop kan verschillen en we vaak niet eens weten hoeveel kinderen een knoop uiteindelijk zal bevatten. De oplossing hiervoor is het gebruik van unranked trees [16]. Bij unranked trees kunnen de knopen een onbepaald aantal kinderen hebben. In deze tekst spreken we enkel van boomautomaten voor unranked trees.

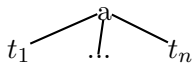
4.2.2 Definitie

Een *niet-deterministische bottom-up boomautomaat (NBTA)* [17] is een tupel $B = (Q, \Sigma, \delta, F)$ met:

- Q: een eindige verzameling toestanden
- Σ: een eindig alfabet
- $F \subseteq Q$: een verzameling eindtoestanden
- δ: een functie $Q \times \Sigma \rightarrow 2^{Q^*}$ zodat $\delta(q, a)$ een reguliere string taal over Q is voor elke $a \in \Sigma$ en $q \in Q$.

De *semantiek* van de automaat B op een boom t, genoteerd als $\delta^*(t)$, is inductief gedefinieerd:

- als t slechts bestaat uit 1 knoop (een blad) met label a dan is $\delta^*(t)$ gelijk aan $\{q|\varepsilon \in \delta(q,a)\}$
- als t een boom is met a in de wortel en $t_1, ..., t_n$ als kinderen (zoals in Figuur 4.1) dan is $\delta^*(t)$ gelijk aan $\{q|\exists q_1 \in \delta^*(t^1), ..., \exists q_n \in \delta^*(t^n)$ en $q_1, ..., q_n \in \delta(q,a)\}$.



Figuur 4.1: De boom t

Een boom t over het alfabet Σ wordt aanvaard door de automaat B als $\delta^*(t) \cap F \neq \emptyset$.

Vervolgens kunnen we zeggen dat een boomautomaat B *deterministisch* is als voor alle $a \in \Sigma$ en $q, q\prime \in Q, q \neq q\prime$ geldt dat $\delta(q,a) \cap \delta(q\prime,a) = \emptyset$.

Net als bij automaten over woorden zijn twee boomautomaten equivalent als ze dezelfde taal aanvaarden. We kunnen dus het volgende noteren:

De boomautomaten $M1$ en $M2$ zijn equivalent als en slechts als $\mathcal{L}(M1) = \mathcal{L}(M2)$.

Uit [19] weten we verder nog dat de taal, aanvaard door een boom, een boomtaal is en dat boomtalen gesloten zijn onder unie, complement en doorsnede.

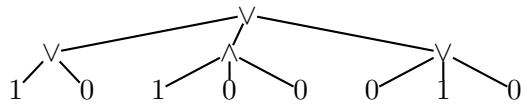
4.2.3 Voorbeeld: booleaanse formules [13]

Beschouwen we het alfabet $\Sigma = \{0, 1, \wedge, \vee\}$ en bomen waarbij 0 en 1 enkel in de bladeren mogen voorkomen en $\wedge$ en $\vee$ enkel in de knopen die geen blad zijn. Deze bomen stellen booleaanse formules voor. We kunnen nu een boomautomaat B definiëren dat exact deze bomen aanvaardt: $B = \{Q, \Sigma, \delta, F\}$ waarbij $Q = \{0, 1\}$, $\Sigma = \{0, 1, \wedge, \vee\}$, $F = \{1\}$ en:

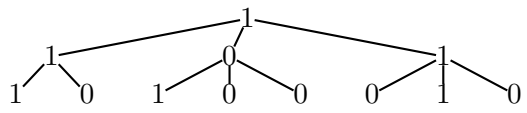
- $\delta(0,0) = \varepsilon$
- $\delta(1,1) = \varepsilon$
- $\delta(0,1) = \emptyset$
- $\delta(1,0) = \emptyset$
- $\delta(1,\wedge) = 1^*$
- $\delta(0,\vee) = 0^*$
- $\delta(0,\wedge) = (0+1)^*0(0+1)^*$
- $\delta(1,\vee) = (0+1)^*1(0+1)^*$

Deze automaat geeft het label 1 (0) aan bladeren met label 1 (0). Een knoop met label $\wedge$ ($\vee$) krijgt het label 1 (0) als alle kinderen van de knoop het label 1 (0) hebben. Deze automaat accepteert een boom als het label van de wortel 1 is. Zie ook Figuur 4.2 en Figuur 4.3.

Voor de duidelijkheid, enkel in de eerste twee transities $\delta(0,0) = \varepsilon$ en $\delta(1,1) = \varepsilon$ worden de 0 en 1 uit het alfabet gebruikt (als tweede parameter), de andere 0-en en 1-en zijn de toestanden 0 en 1.



Figuur 4.2: Een boom onder het alfabet Σ van 4.2.3



Figuur 4.3: Een accepterende run van de automaat van 4.2.3

4.3 Toepassing XML en DTD

4.3.1 XML: eXtensible Markup Language

XML of eXtensible Markup Language [11, 12] is een eenvoudig tekstformaat afgeleid van SGML (ISO 8879). XML verbindt tekst met de betekenis (structuur en soort inhoud) ervan. De betekenis van tekst wordt weergegeven in tags, vergelijkbaar met HTML. Een tag bestaat uit twee delen: een start-tag bijvoorbeeld `<titel>` en een eind-tag `</titel>`. Zo kunnen we de titel van deze paragraaf als volgt beschrijven:

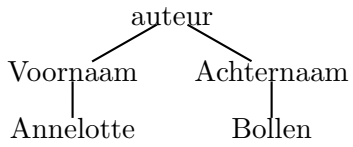
```
<titel>XML: eXtensible Markup Language</titel>
```

Als er niets tussen de tags komt te staan, kunnen we de twee tags combineren: `<titel/>`. Deze tags worden element-tags genoemd en ze definiëren de markup. Er zijn geen standaard tags, dus mogen we zelf de naam en betekenis van een tag kiezen. Net zoals bij HTML kunnen tags genest worden. Bijvoorbeeld:

```
<?XML version="1.0"?>
<auteur>
    <voornaam>Annelotte</voornaam>
    <achternaam>Bollen</achternaam>
</auteur>
```

De eerste regel `<?XML version="1.0"?>` moet steeds bovenaan in het document voorkomen, anders heb je geen geldig XML document.

We kunnen een XML document ook voorstellen als een boom. De tekst tussen de tags komt dan telkens in de bladeren van de boom. De andere knopen of nodes zijn de element-tags. De boom van het bovenstaande voorbeeld kan bekeken worden in figuur 4.4.



Figuur 4.4: Voorbeeld van een XML document voorgesteld als boom

Element-tags kunnen ook attributen bevatten, een beetje zoals argumenten aan een functie kunnen doorgegeven worden. Attributen specifiëren we binnen de tag, bijvoorbeeld:

```
<auteur geslacht="vrouw"> ... </auteur>
```

We moeten zelf bepalen of iets als attribuut meegegeven wordt of dat we het tussen element-tags willen zetten. Attributen gebruiken we normaal gezien enkel voor informatie die we aan de element-tag willen toevoegen maar die we niet expliciet willen tonen aan de gebruiker.

In XML kunnen we ook macro's definiëren voor veelgebruikte tekst of verwijzen naar andere stukken tekst. Dit doen we door een entity te definiëren. Een entity reference of verwijzing begint steeds met een ampersand (&) en eindigt met een puntkomma (;). Als de volgende entity gedefinieerd is, kunnen we ernaar verwijzen met: `&luc;`. Hierdoor wordt "`&luc;`" vervangen door "Limburgs Universitair Centrum".

```
<!ENTITY luc "Limburgs Universitair Centrum">
```

Om commentaar op te nemen in een XML document, plaatsen we dit tussen `<!-- en -->`. Verder kan een XML-bestand ook processing instructies, CDATA secties en document type declarations (DTD) bevatten.

Een goed gestructureerd (*well-formed*) XML document moet aan de volgende vereisten voldoen:

- De resulterende boom heeft slechts een wortel; er is dus een unieke element-tag die als root fungeert.
- De tags moeten gebalanceerd zijn. Dit wil zeggen dat er voor elke start-tag een eind-tag moet zijn.
- De tags moeten goed genest zijn, net zoals haakjes bij wiskundige formules.
- Attributen (zowel tekst als getallen) moeten tussen quotes staan zoals:
  `<auteur geslacht="vrouw"> ... <\auteur>`

- De gereserveerde tekens >, < en & mogen niet gebruikt worden in gewone tekst of attributen, indien we dit wel wensen, werken met een CDATA sectie, daar wordt echter niet dieper op ingegaan.
- Er zijn vijf voorgedefinieerde referenties:
  - `&copy;` dit geeft ©
  - `&reg;` dit geeft ®
  - `&tm;` dit geeft ^{TM}
  - `&eacute;` dit geeft é
  - ` ` dit geeft (een spatie dus)

4.3.2 Equivalentie van XML

Indien we werken met een grote database van XML-bestanden moet er een mogelijkheid zijn om na te gaan of twee bestanden al dan niet equivalent zijn. Het kan namelijk gebeuren dat bepaalde systemen de volgorde van attributen of de structuur van een entiteit aanpassen zonder dat daardoor de inhoud verandert. De bestanden zijn dus niet identiek maar stellen wel dezelfde data voor. Voor dit doeleinde is er een specificatie ontwikkeld door de XML Signature werkgroep van het W3C consortium [22].

Deze specificatie beschrijft een methode om een fysische voorstelling te maken van een XML-bestand; de canonische vorm ofwel canonische XML. Als twee XML-bestanden dezelfde canonische vorm hebben, zijn ze *logisch equivalent*. We moeten er wel rekening mee houden dat er enkel een als-dan relatie beschreven wordt en geen als-en-slechts-als. Het kan dus voorkomen dat bestanden wel equivalent zijn maar niet dezelfde canonische vorm hebben. Dit komt door applicatiespecifieke wijzigingen die kunnen voorkomen maar niet veralgemeend kunnen worden. Denk hierbij bijvoorbeeld aan de voorstelling van een kleur: <color>black</color> en <color>rgb(0,0,0)</color>. Canonische XML-bestanden zijn nog steeds well formed XML-bestanden en kunnen ook als boom voorgesteld worden.

Indien we willen testen of een XML-bestand een ander bestand omvat, zullen we ook eerst de bestanden in hun canonische vorm noteren. Vervolgens stellen we ze voor als boom en controleren we of de ene boom al dan niet een subtree is van de andere. De subtree bepaalt in dat geval het XML-bestand dat volledig beschreven wordt door het andere.

De details van canonisatie voor de geïnteresseerde lezer staan op [22], als illustratie het voorbeeld met volgend XML-bestand:

```
<?xml version="1.0"?>
<book language="english"      code="Sip1997"> <!-- comment -->
  <date    ></date>


  <title> Introduction to the theory of computation </title>
  <authors>
    <author>
     <name> Sipser </name>
                                  <first_name> Michael </first_name>
    </author>
  </authors>

  <keywords> computation automata </keywords>
</book>
```

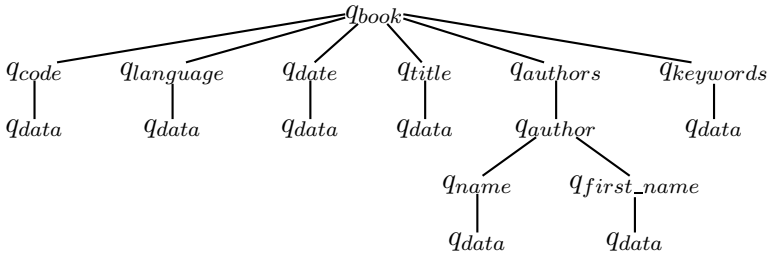
De canonische vorm van dit bestand is:

```
<book code="Sip1997" language="english">
  <date></date>
  <title> Introduction to the theory of computation </title>
  <authors>
    <author>
     <name> Sipser </name>
     <first_name> Michael </first_name>
    </author>
  </authors>
  <keywords> computation automata </keywords>
</book>
```

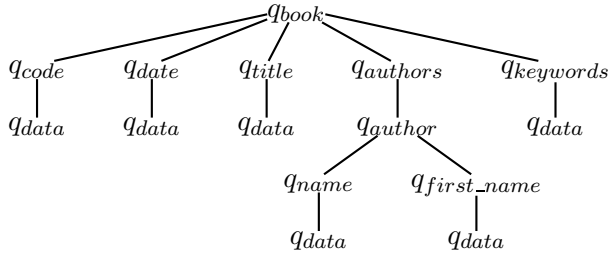
Merk op dat de XML declaratie, commentaar en nutteloze whitespaces verdwenen zijn. Men kan ook opteren de commentaar te behouden. Dit noemt men de commented canonical form. Verder worden attributen (en eventuele namespaces) alfabetisch opgesomd. Het volgende XML-bestand heeft een andere canonische vorm (het attribuut language is namelijk verdwenen) en is dus niet equivalent met het eerste:

```
<?xml version="1.0"?>
<book code="Sip1997"> <!-- comment -->
  <date   />
  <title> Introduction to the theory of computation </title>
  <authors>
    <author>
     <name> Sipser </name>
     <first_name> Michael </first_name>
    </author>
  </authors>
  <keywords> computation automata </keywords>
</book>
```

Als we van beide canonische voorstellingen de boom bekijken (zie figuren 4.5 en 4.6), merken we dat de tweede boom een subtree is van de eerste. Hierdoor kunnen we dan ook besluiten dat het eerste XML-bestand het tweede omvat. Het tweede bestand brengt ons geen nieuwe of gewijzigde informatie aan.



Figuur 4.5: Boomvoorstelling van het eerste XML-bestand



Figuur 4.6: Boomvoorstelling van het tweede XML-bestand

4.3.3 DTD's

We kunnen regels opleggen waaraan een XML-bestand moet voldoen. In de vorige paragraaf bespraken we reeds de goed gestructureerde of *well-formed* bestanden. Nu gaan we een stap verder. Aan de hand van een DTD of Document Type Definition [12] gaan we extra regels opleggen waaraan een XML document moet voldoen. Het nagaan of een document aan de bijbehorende DTD voldoet, noemen we valideren. De DTD kan opgenomen zijn in het XML-bestand of in een afzonderlijk bestand staan waarnaar verwezen wordt in het XML-bestand.

Een DTD legt beperkingen op voor:

- Element namen
- Attribuut namen en waarden
- De volgorde van elementen
- De nesting van elementen

Elementen worden gedefinieerd met volgende syntax:

```
<!ELEMENT element_name content_model>
```

"element_name" Verwijst naar de naam van de element-tag die je wilt definiëren en de mogelijke vorm van de subtree onder het element wordt bepaald door "content_model". Dit kan gewone tekst zijn (textual content: `#PCDATA, EMPTY of ANY`) of een gecombineerde inhoud hebben zoals tekst, namen van elementen die onder dit element kunnen voorkomen, namen van parameters en combinaties hiervan. Deze gecombineerde inhoud kan men uitdrukken als (cfr. reguliere expressies[1]):

- E
  Er moet exact 1 keer het element E voorkomen in de subtree (1 keer)
- E?
  Er kan maximaal 1 keer het element E voorkomen in de subtree (0 of 1 keer)
- E+
  Er moet minstens 1 keer het element E voorkomen in de subtree (1 of meer keer)
- E*
  E kan voorkomen, en het aantal keer is onbepaald in de subtree (0 of meer keer)
- E1|E2
  E1 of E2 moet voorkomen in de subtree
- E1,E2
  E1 en E2 moeten in deze volgorde voorkomen in de subtree (het zijn dus siblings)

Als E1 of E2 van het type `#PCDATA` is, mag men geen AND combinatie maken (E1,E2) en bij een OR combinatie (E1|E2) moet men er steeds voor zorgen dat het element met `#PCDATA`-type eerst komt (E1 in dit geval).
De syntax voor attributen ziet eruit als:

```
<!ATTLIST element_name attribute_name values default_value>
```

Element_name verwijst naar het element waarvoor we de parameters willen bepalen. Verder wordt de naam van het attribuut, de mogelijke waarden en eventueel een default waarde gedefinieerd. De laatste drie delen mogen herhaald worden indien een element meerdere attributen heeft. De mogelijke types voor attributen zijn:

- CDATA
  String zonder < en zonder ".
- ID
  Unieke ID.
- IDREF
  Een referentie naar een ID.

[1] content_ model wordt bepaald door een reguliere expressie

- IDREFS
  Een serie referenties, gescheiden door spaties.
- NMTOKEN
  Een legale naamtoken; dit is een geldige XML naam.
- NMTOKENS
  Een serie naamtokens, gescheiden door spaties.
- ENTITY
  Een entity die gedeclareerd is in de DTD.
- ENTITIES
  Een serie entities, gescheiden door spaties.
- NOTATION
  Een notatie type dat gedeclareerd is in de DTD, dit formaat wordt niet door de XML processor verwerkt.
- Opsomming
  Een opsomming van mogelijke waarden, door de gebruiker gedefinieerd.

We kunnen attributen op vier verschillende manieren declareren:

- `#REQUIRED`
  Er moet een waarde aan het attribuut gegeven worden.
- `#IMPLIED`
  De XML processor moet expliciet aan de applicatie vermelden dat er geen waarde is ingevuld.
- `#FIXED`
  De waarde die aan het attribuut toegekend wordt, moet de default waarde zijn.
- Default waarde
  Als er een default waarde gegeven is, zal deze gebruikt worden als het attribuut niet gebruikt werd in de desbetreffende tag. Anders wordt de opgegeven waarde gebruikt.

Uitgewerkt voorbeeld

Hieronder staat een voorbeeld van een DTD die informatie over een boek bevat. Elk boek heeft een ID die als attribuut gedefinieerd is. Een boek kan meerdere auteurs hebben maar slechts één datum en titel. Ook is er een element `keywords` opgenomen waarin een willekeurige string van sleutelwoorden kan staan. De sleutelwoorden in deze DTD worden echter steeds in zijn geheel beschouwd en niet woord per woord. Indien men de sleutelwoorden apart wil opslaan zou men `keywords` moeten definiëren zoals `authors`. Ook moet een nieuw element `<!ELEMENT keyword (#PCDATA)>` gedefinieerd worden.

```
<!DOCTYPE book [
 <!ELEMENT book (date, title, authors, keywords)>
  <!ATTLIST book code ID #REQUIRED>
  <!ELEMENT date (#PCDATA)>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT authors (author+)>
    <!ELEMENT author (name, first_name)>
    <!ELEMENT name (#PCDATA)>
    <!ELEMENT first_name (#PCDATA)>
  <!ELEMENT keywords (#PCDATA)*>
]>
```

Het onderstaande XML document is een voorbeeld van een geldig XML document voor bovenstaande DTD.

```
<?xml version="1.0"?>
<book code="Sip1997">
  <date> 1997 </date>
  <title> Introduction to the theory of computation </title>
  <authors>
    <author>
     <name> Sipser </name>
     <first_name> Michael </first_name>
    </author>
  </authors>
  <keywords> computation automata </keywords>
</book>
```

Beide voorbeelden zijn uitgelijnd ter verduidelijking voor de lezer. Newlines en spaties worden echter genegeerd door programma's die met XML en DTD werken.

4.3.4 DTD als boomautomaat

Als we teruggaan naar de eerder besproken XML-bestanden kunnen we zeggen dat de DTD de transitieregels van de boomautomaat bepaalt en dat we kunnen testen of een XML-bestand aan een DTD voldoet door de automaat erop los te laten. Het is te bewijzen [13] dat elke DTD uitdrukbaar is door middel van een boomautomaat.

Het voorstellen van een XML-bestand als boom is reeds besproken in sectie 4.3.1. Er werd echter nog niet gesproken over attributen. Willen we een XML-bestand met attributen in de elementen voorstellen als boom, dan kunnen we op verschillende manieren te werk gaan. De makkelijkste manier, die ook voor dit werk gebruikt wordt, is het behandelen van een attribuut als een element. Aangezien ontwerpers vrij zijn om te bepalen of ze iets als attribuut of als element meegeven, houdt dit geen verlies van functionaliteit in. Een andere manier was elke element-knoop twee kinderen te geven: één met alle attributen en één met alle elementen. Hierdoor maken we de boom natuurlijk weer een stap dieper. Referenties worden voorlopig genegeerd.

We gaan zo te werk: beschouw een DTD S van de vorm:

$$<!\text{ELEMENT } l_1\ c_1>$$
$$<!\text{ELEMENT } l_2\ c_2>$$
$$\ldots$$
$$<!\text{ELEMENT } l_k\ c_k>$$

Hierbij zijn $l_1, ..., l_k$ labels en $c_1, ..., c_k$ reguliere expressies over de verzameling van vermelde labels plus de datatypes (deze worden algemeen genoteerd als "data").

We willen nu een boomautomaat $\Lambda_S = (Q, \delta, F)$ construeren over het alfabet $\Sigma_S = \{l_1, ..., l_k, data\}$ die alle geldige XML-bestanden aanvaardt.

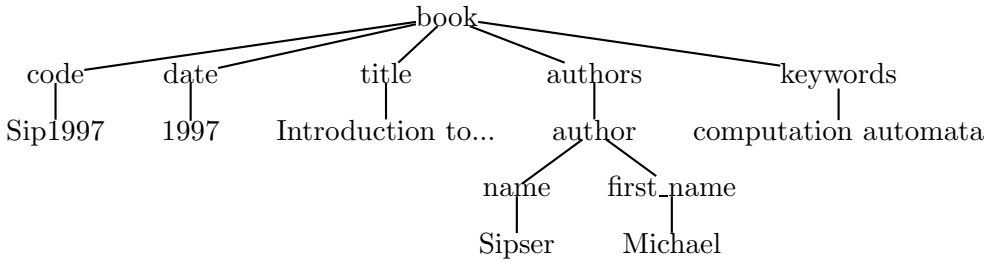
1. In de verzameling van alle toestanden Q plaatsen we een toestand q_x voor alle $x \in \Sigma_S$
2. F = Q, tenzij we weten wat de wortel van het bestand is, gebruiken we de toestand die naar de wortel verwijst.
3. voor elke declaratie <!ELEMENT l c> voegen we een transitie $\delta(q_l, l) = c'$ toe met c' de reguliere expressie c waarbij alle labels x vervangen zijn door de respectieve toestanden q_x. Let op, in c moet elk datatype vervangen zijn door $data$, in c' wordt dit dan q_{data}.
4. We voegen een transitie $l(q_{data}, data) = \epsilon$ toe.
5. Alle andere transities $l(q_a, b)$ met $a \neq b$ zijn leeg.

Uitgewerkt voorbeeld

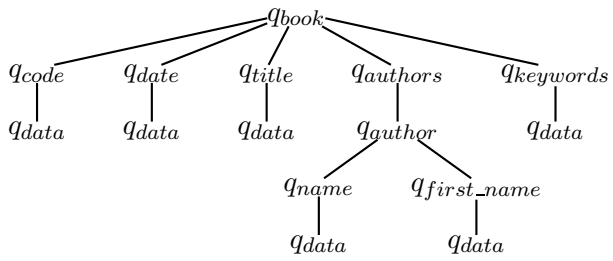
We gaan de DTD van 4.3.3 omzetten in een boomautomaat $B = (Q, \Sigma, \delta, F)$. Denk eraan dat attributen als elementen behandeld worden!

- $Q = \{q_{book}, q_{code}, q_{date}, q_{title}, q_{authors}, q_{author}, q_{name}, q_{first\_name}, q_{keywords}, q_{data}\}$
- $\Sigma = \{$book, code, date, title, authors, author, name, first_name, keywords, $data\}$
- $F = \{q_{book}\}$
- $\delta =$ (alle niet vermelde transities zijn $\emptyset$)
  - $\delta(q_{book}, book) = q_{code}q_{date}q_{title}q_{authors}q_{keywords}$
  - $\delta(q_{code}, code) = q_{data}$
  - $\delta(q_{date}, date) = q_{data}$
  - $\delta(q_{title}, title) = q_{data}$
  - $\delta(q_{authors}, authors) = q_{author}+$
  - $\delta(q_{author}, author) = q_{name}q_{first\_name}$
  - $\delta(q_{name}, name) = q_{data}$
  - $\delta(q_{first\_name}, first\_name) = q_{data}$
  - $\delta(q_{keywords}, keywords) = q_{data}*$
  - $\delta(q_{data}, data) = \{\varepsilon\}$

Bemerk dat niet alleen q_{data} toestand kan zijn van een blad, maar ook $q_{keywords}$ omwille van de $*$ in de reguliere expressie aan de rechterkant. Deze automaat aanvaardt een boom als de toestand van de wortel gelijk is aan q_{book}. Het is duidelijk dat elke DTD uitdrukbaar is door middel van een boomautomaat.
In figuur 4.7 ziet u het XML-bestand van 4.3.3 én figuur 4.8 laat een geldige run zien van onze automaat op deze boom.



Figuur 4.7: De boomstructuur van het XML-bestand in 4.3.3



Figuur 4.8: Geldige run van een automaat van de DTD in 4.3.3

4.3.5 Ondubbelzinnig gedefinieerde DTD's

Voor compatibiliteit met SGML wordt in de XML specificatie [11] nog een extra eis gesteld aan DTD's: het is noodzakelijk dat DTD's *deterministisch* gedefinieerd worden. Deterministische DTD's worden bepaald door deterministische of niet-ambigue reguliere expressies.

Een taal kan op verschillende manieren voorgesteld worden met een reguliere expressie maar een reguliere expressie bepaalt slechts één taal. Indien we een woord willen parsen volgens een reguliere expressie, moeten we weten met welk symbool van de expressie een symbool uit het woord overeenkomt. Beschouw bijvoorbeeld de expressie $(a + b)^*aa^*$. Indien we een woord parsen en we stellen dat een bepaalde letter a uit het woord overeenkomt met de tweede a in de expressie, weten we dat er geen enkele b meer mag voorkomen in het woord. Maar hoe bepalen we juist met welke a uit de expressie een a in het woord overeenkomt? Deze problematiek wordt besproken door Brüggemann–Klein en Wood in [8].

In het artikel worden symbolen op twee manieren gematcht, met of zonder vooruit te kijken naar de rest van het woord. Als we elk symbool op slechts één manier kunnen matchen, spreken we van een *niet-ambigue (unambiguous)* reguliere expressie. Indien er meerdere mogelijkheden zijn om een symbool te matchen, spreken we van een *ambigue (ambiguous)* reguliere expressie. Indien we kunnen matchen zonder naar de rest van het woord te kijken, dus één symbool per keer bekijken en niet vooruit kijken, wordt er gesproken van *one-unambiguous* reguliere expressies.

Bijvoorbeeld: [8]

- $(a + b)^*aa^*$ is ambigu want:
  geef elk symbool een positie als subscript zodat de verschillen duidelijk zijn: $(a_1+b)^*a_2a_3^*$
  beschouw het woord aaa, dit woord kan gematcht worden als: $a_1a_1a_2$, $a_1a_2a_3$ en als $a_1a_3a_3$
  dus op drie verschillende manieren
- $(a_1 + b)^*a_2$ is niet ambigu maar men moet vooruit kunnen kijken, deze expressie is dus niet one-unambiguous
- $b^*a(b^*a)^*$ is one-unambiguous

Merk op dat deze drie reguliere expressies steeds dezelfde taal beschrijven.

Brüggemann–Klein en Wood gebruiken de definitie uit the Standard Generalized Markup Language (SGML) als basis voor hun definitie van one-unambiguous expressions. Deze definitie houden we aan, aangezien we net op zoek zijn naar compatibiliteit tussen DTD's en SGML.

Een expressie E is one-unambiguous als en slechts als, voor alle woorden u, v, w over Π en alle symbolen $x, y \in \Pi$, geldt dat als $uxv, uyw \in L(E')$ en $x \neq y$ dan $x^t \neq y^t$. Een *taal* is one-unambiguous als er een one-unambiguous expressie bestaat die deze taal beschrijft.

Waarbij Π de verzameling is van elementen van het alfabet waaraan een subscript is toegevoegd. Het toevoegen van subscripts aan een expressie noemen we markeren of marking, E' is de gemarkeerde versie van E waarbij elk gemarkeerd symbool maximaal één keer voorkomt. Het omgekeerde van markeren is het laten vallen van markeringen, notatie E^t, bijvoorbeeld $x_1^t = x$. $L(E')$ is de taal voortgebracht door de gemarkeerde reguliere expressie E'.

In hoofdstuk 3 over stringautomaten en reguliere expressies werden twee methodes besproken om van een reguliere expressie een eindige automaat te maken. Eén van deze methodes was de Glushkov-constructie waarbij gebruik gemaakt werd van de drie verzamelingen $first(L), last(L)$ en $follow(L, a)$. Gebruiken we deze verzamelingen zoals gedefinieerd in 3.3 op pagina 26, dan zijn volgende statements eenvoudig na te gaan. [8]

Voor elke gemarkeerde expressie E een woord $x_1...x_n$ over Π tot de taal van E behoort als en slechts als:

- $x_1 \in first(E)$
- $x_n \in last(E)$
- $x_{i+1} \in follow(E, x_i)$ voor alle i tussen 1 en n

Een expressie E is dan one-unambiguous als en slechts als:

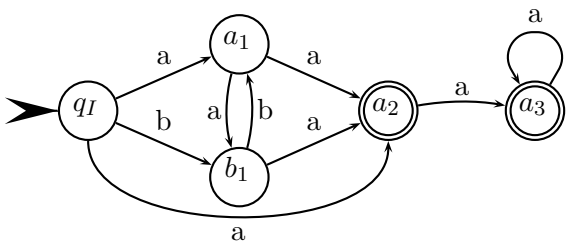
- $\forall x, y \in first(E')$: als $x \neq y$ dan $x^t \neq x^t$ en
- $\forall z$ symbool van E' en $x, y \in follow(E', z)$: als $x \neq y$ dan $x^t \neq x^t$

Als we dit laatste statement langs de constructie van een Glushkov-automaat leggen, kunnen we afleiden dat een reguliere expressie one-unambiguous is als en slechts als de Glushkov-automaat deterministisch is.

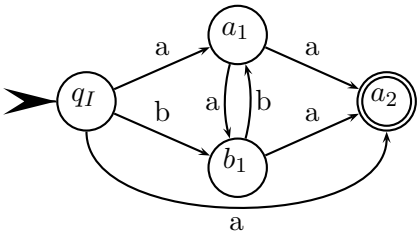
Voorbeeld

Beschouwen we de Glushkov-automaten van volgende reguliere expressies:

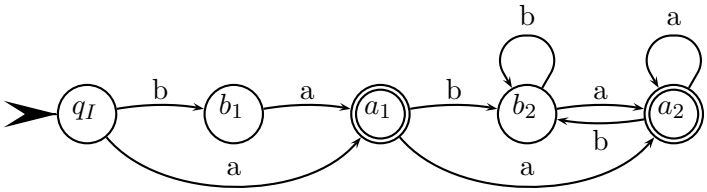
- $(a+b)^*aa^*$ is ambigu, de Glushkov-automaat in figuur 4.9 is duidelijk niet-deterministisch, vanuit enkele toestanden vertrekken meerdere pijlen met hetzelfde label.
- $(a_1 + b)^*a_2$ is niet ambigu maar ook niet one-unambiguous, figuur 4.10, vanuit toestand a_1 vertrekken twee pijlen met label a.
- $b^*a(b^*a)^*$ is one-unambiguous, zie figuur 4.11, hierin ziet u een deterministische automaat (vanuit elke toestand vertrekken juist twee pijlen, een met label a en een met label b.



Figuur 4.9: Glushkov-automaat voor $(a_1 + b_1)^*a_2a_3^*$



Figuur 4.10: Glushkov-automaat voor $(a_1 + b_1)^* a_2$



Figuur 4.11: Glushkov-automaat voor $b_1^* a_1 (b_2^* a_2)^*$

4.4 Toepassing XPath

4.4.1 Standaard XPath

XPath [12] is een querytaal om delen van een XML-bestand zoals elementen en attributen te vinden. Zo kan men met behulp van XPath bijvoorbeeld alle boeken opvragen van een bepaalde auteur in een XML-bestand dat bibliotheekgegevens bevat. Voor XPath expressies te lezen, beschouwen we het XML-bestand als een boom. We navigeren steeds vanuit een context; dat wil zeggen we beginnen bij een bepaalde knoop en het gevonden antwoord hangt ook steeds af van het vertrekpunt. Een XPath expressie wordt als volgt opgebouwd:

1. Bepaal de richting (axis) van navigatie door de boom in relatie tot de huidige positie. Hierbij kan er gekozen worden uit:

   - self: dit staat voor de huidige knoop
   - child: al de rechtstreekse kinderen
   - descendant: alle afstammelingen
   - parent: de knoop waar deze knoop een rechtstreeks kind van is
   - ancestor: alle voorouders
   - attribute: een attribuut van de huidige knoop
   - namespace: selecteert alle namespace knopen van de huidige knoop
   - following-sibling: selecteert alle andere kinderen van de ouder van deze knoop die na de huidige knoop komen
   - preceding-sibling: alle siblings van deze knoop die hem voorafgaan

- following: alle knopen uit het document die na deze knoop komen en er geen afstammeling van zijn
- preceding: alle voorgaande knopen maar niet de voorouders
- descendant-or-self: de huidige knoop en al zijn afstammelingen
- ancestor-or-self: de huidige knoop en zijn voorouders

Voor de duidelijkheid: ancestor, descendant, following, preceding en self zijn disjuncte verzamelingen van knopen en samen bevatten zij alle knopen uit het bestand.

2. Bepaal het type of de naam van knoop die we zoeken (node test). Hiervoor kan gekozen worden uit $*$ voor alle knopen, E voor een element met de naam E, $node()$ voor knopen, $text()$ voor tekst elementen, $comment()$ voor commentaren en $processing-instruction()$.

3. Vraag uiteindelijk, op basis van de inhoud of andere eigenschappen van de verzameling nodes, wat we willen weten (predicate). Dit wordt gedaan met een booleaanse of numerieke expressie tussen hoekhaken. Ook voor dit onderdeel van een XPath expressie bestaan er een aantal voorgedefinieerde functies. Hier worden er enkele opgenoemd:

   - count(node-set): geeft het aantal knopen in de verzameling
   - contains(string,string): bevat de eerste string de tweede string?
   - string(object): geeft een string weer die het object voorstelt

Samengevoegd gebruiken we de volgende syntaxis: $axis :: Nodetest[predicate]$

XPath expressies kunnen al snel lange uitdrukkingen worden. Daarom werden er een reeks afkortingen ingevoerd. Zo mag men bijvoorbeeld de axis "child::" gewoon weglaten, "descendant-or-self::" wordt vervangen door "//", "attribute" door "@", "self::node()" wordt "." en "parent::node()" wordt "..".

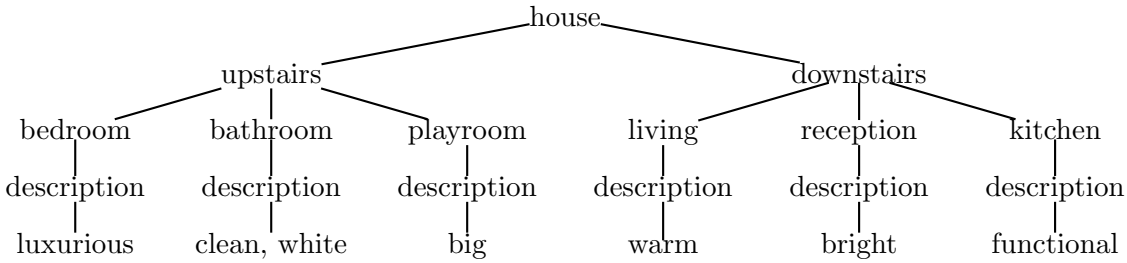
Uitgewerkt voorbeeld

Beschouwen we het XML-bestand in figuur 4.12, dan kunnen we bijvoorbeeld de volgende XPath expressies schrijven:

- $//*[following-sibling :: kitchen]$
  selecteert alle knopen (living en reception in dit geval) die vóór de keuken staan in het bestand. Als eerste bekijken we de axis $(descendant-or-self)$; als we de wortel van de boom als context node nemen zou dit betekenen dat we tussen alle knopen zoeken. Daardoor is // waarschijnlijk de meest gebruikte axis. De node test is $*$; hierdoor worden alle soorten knopen bekeken. Het predikaat $following-sibling :: kitchen$ bepaalt uiteindelijk de voorwaarden aan welke de knoop moet voldoen om als resultaat behouden te worden: één van de volgende broers van de knoop moet *kitchen* zijn.

- $count(//*[parent :: upstairs])$
  telt het aantal kamers op de bovenverdieping ofwel het aantal knopen dat als ouder upstairs heeft.

- */upstairs/∗*
  geeft alle kinderen van de knoop upstairs als we beginnen bij de wortel. Indien je hier de *count*() functie rondzet geeft deze hetzelfde resultaat als het vorige voorbeeld.
- *// ∗ [description = "luxurious"]*
  geeft alle knopen terug die een description knoop hebben met de tekst luxurious.
- *//description[parent :: bathroom]* geeft de beschrijving van de badkamer. De node test *description* bepaalt hier dat we tussen alle description knopen zoeken.



Figuur 4.12: Een XML-bestand

4.4.2 Core XPath

Omdat het bestuderen van alle aspecten van XPath een voltijdse bezigheid zou zijn, beperken veel artikels zich tot CORE XPath [13, 14]; een abstractie van de ondervragingstaal die enkel de kern van de zaak behandelt. Hierbij laten we de verschillende functieaanroepen en berekeningen achterwege en baseren we ons op de asbewegingen en filters.

Syntax en semantiek

De syntax van Core XPath is gedefinieerd door de volgende grammatica:

- *Core XPath Expressie* $\kappa ::= \pi \mid /\pi \mid \kappa|\kappa$
- *pad expressie* $\pi ::= \alpha :: \theta \mid \alpha :: \theta[\epsilon] \mid \pi/\pi$
- *asbewegingen* $\alpha ::= self \mid \beta|\beta^* \mid \beta^+$
- *basisassen* $\beta ::= \Leftarrow|\Rightarrow|\Downarrow|\Uparrow$
- *labeltests* $\theta ::= \star \mid l_1 \mid ... \mid l_k$
- *filterexpressies* $\epsilon ::= \kappa \mid$ not $\epsilon \mid \epsilon$ and $\epsilon \mid \epsilon$ or ϵ

Hierbij veronderstellen we dat $\Lambda = l_1, ..., l_k$ en de asbewegingen genoteerd worden zoals in tabel 4.1.

Core XPath	XPath
$\Uparrow^*$	ancestor-or-self
$\Uparrow^+$	ancestor
$\Uparrow$	parent
$\Leftarrow^* \Leftarrow^+ \Leftarrow$ self $\Rightarrow \Rightarrow^+ \Rightarrow^*$	preceding-sibling self following-sibling
$\Downarrow$	child
$\Downarrow^+$	descendant
$\Downarrow^*$	descendant-or-self

Tabel 4.1: Notatie voor de axis

Aangezien we werken op de boomstructuur van XML-bestanden geven we eerst de definitie van een boom: Een boom is een tupel (N_*, E, S, L) waarbij:

- N_* de verzameling van knopen is;
- $E \subseteq N_* \times N_*$ de ouder-van relatie tussen de knopen uitdrukt;
- $S \subseteq N_* \times N_*$ de linker-broer-van relatie uitdrukt;
- $L : N_* \to \Lambda$ de labelingsfunctie is die elke knoop met zijn label associeert.//

Net zoals een gewone XPath expressie selecteert Core XPath knopen in de boom vertrekkende van een context knoop. Hierdoor kunnen we stellen dat een Core XPath expressie een binaire relatie $\kappa(t)$ definieert op de knopen van een boom t, waarbij $(n, n') \in \kappa(t)$ als en slechts als n' geselecteerd wordt door de expressie κ vanuit n.

Filterexpressies in een expressie resulteren in een booleaanse waarde; ze geven *true* in een bepaalde knoop als we er knopen uit kunnen bereiken die aan de filter voldoen, zoniet wordt $false$ gegeven. We gebruiken de volgende notatie voor een filterexpressie c die *true* is in knoop n van boom t: $(t, n) \models c$.

De semantiek van Core XPath kunnen we dan als volgt definiëren: [13]
Zij $t = (N, E, S, L)$ een boom. We schrijven N_l voor de verzameling knopen van t die gelabeld zijn met l. Vervolgens schrijven we R^+ voor de transitieve sluiting van een binaire relatie R, R^* voor de reflexief-transitieve sluiting van R en R^{-1} voor het invers van R. De semantiek is dan inductief gedefinieerd als:

1. *Evaluatie van Core XPath en pad expressies*
   $(\alpha :: \theta)(t) := \{(n, n') | (n, n') \in \alpha(t) \text{ en } n' \in N_\theta\}$
   $(\alpha :: \theta[\epsilon])(t) := \{(n, n') | (n, n') \in \alpha(t),\ n' \in N_\theta \text{ en } (t, n') \models \epsilon\}$
   $(\pi_1/\pi_2)(t) := \{(n, n'') | \exists n' : (n, n') \in \pi_1(t) \text{ en } (n', n'') \in \pi_2(t)\}$
   $(/\pi)(t) := \{(root, n') | (root, n') \in \pi(t)\}$
   $(\kappa_1 | \kappa_2)(t) := \kappa_1(t) \cup \kappa_2(t)$

2. *Evaluatie van asbewegingen en basisassen*

   $self(t) := \{(n, n) | n \in N_l\}$
   $\Rightarrow (t) := S$ $\quad\quad$ $\Leftarrow (t) := S^{-1}$
   $\Downarrow (t) := E$ $\quad\quad$ $\Uparrow (t) := E^{-1}$
   $(\beta^+)(t) := \beta(t)^+$ $\quad\quad$ $(\beta^*)(t) := \beta(t)^*$

3. *Evaluatie van filterexpressies*
   $(t, n) \models \pi \quad \Leftrightarrow \quad \exists n' : (n, n') \in \pi(t)$
   $(t, n) \models \epsilon_1 \text{ and } \epsilon_2 \quad \Leftrightarrow \quad (t, n) \models \epsilon_1 \text{ en } (t, n) \models \epsilon_2$
   $(t, n) \models c_1 \text{ or } c_2 \quad \Leftrightarrow \quad (t, n) \models c_1 \text{ of } (t, n) \models c_2$
   $(t, n) \models \text{not } c \quad \Leftrightarrow \quad (t, n) \not\models c$

Voorbeelden van Core XPath expressies

Als we de boom van 4.4.1 beschouwen, kunnen we volgende Core XPath expressies formuleren met (voor zover mogelijk) dezelfde betekenis als voordien:

- $\Downarrow^*:: *[\Rightarrow^+:: kitchen]$
- $\Downarrow^*:: *[\Uparrow:: upstairs]$
  De functie count bestaat niet in Core XPath!
- $/ \Downarrow:: upstairs / \Downarrow:: *$
- $\Downarrow^*:: *[description = "luxurious"]$
  Deze expressie kunnen we niet naar Core XPath vertalen omdat we de gegevens van de description knopen hebben weggelaten.
- $\Downarrow^*:: description[\Uparrow^+:: bathroom]$

Evaluatie van een Core XPath expressie

Willen we nu de eerste expressie evalueren, dan zullen we de boom uit 4.4.1 definiëren als een tupel $t = (N, E, S, L)$. Voor de leesbaarheid zullen we de bladeren van de boom achterwege laten. De verzamelingen N, E, S en L worden dan:

- $N = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$
- $E = \{(1, 2), (1, 3), (2, 4), (2, 5), (2, 6), (3, 7), (3, 8), (3, 9), (4, 10), (5, 11), (6, 12), (7, 13), (8, 14), (9, 15)\}$
- $S = \{(2, 3), (4, 5), (5, 6), (7, 8), (8, 9)\}$
- $L = \{(1, house), (2, upstairs), (3, downstairs), (4, bedroom), (5, bathroom), (6, playroom), (7, living), (8, reception), (9, kitchen), (10, description), (11, description), (12, description), (13, description), (14, description), (15, description)\}$

Vervolgens zullen we $\Downarrow^*:: *[\Rightarrow^+:: kitchen]$ stap voor stap evalueren op t:

$(\Downarrow^* :: *[\Rightarrow^+ :: kitchen])(t)$
$\quad \Updownarrow \Rightarrow^+ :: kitchen$ is van de vorm $\alpha :: \theta$ met $\alpha = \Rightarrow^+$ en $\theta = kitchen$,
$\quad \Updownarrow$ noem deze expressie ϵ
$(\Downarrow^* :: *[\epsilon])(t)$
$\quad \Updownarrow$ hierin herkennen we $(\alpha :: \theta[\epsilon])(t)$ met $\alpha = \Downarrow^*$ en $\theta = *$
$\quad \Updownarrow$ deze expressie evalueren we met behulp van de tweede evaluatieregel:
$\quad \Updownarrow (\alpha :: \theta[\epsilon])(t) := \{(n, n') | (n, n') \in \alpha(t),\ n' \in N_\theta$ en $(t, n') \models \epsilon\}$
$\{(n, n') | (n, n') \in \Downarrow^* (t)$ en $n' \in N_*$ en $(t, n') \models \epsilon\}$
$\quad \Updownarrow$ weglaten van $n' \in N_*$ en evaluatie van $\Downarrow^*$
$\{(n, n') | (n, n') \in E^*(t)$ en $(t, n') \models \epsilon\}$
$\quad \Updownarrow$ evaluatie van ϵ met behulp van de eerste evaluatieregel:
$\quad \Updownarrow (\alpha :: \theta)(t) := \{(n, n') | (n, n') \in \alpha(t)$ en $n' \in N_\theta\}$
$\{(n, n') | (n, n') \in E^*(t)$ en $\exists n'' : (n', n'') \in \{(m, m') | (m, m') \in \Rightarrow^+ (t)$ en $m' \in N_{kitchen}\}\}$
$\quad \Updownarrow$ evaluatie van $\Rightarrow^+$
$\{(n, n') | (n, n') \in E^*(t)$ en $\exists n'' : (n', n'') \in \{(m, m') | (m, m') \in S^+(t)$ en $m' \in N_{kitchen}\}\}$
$\quad \Updownarrow$ vereenvoudigen
$\{(n, n') | (n, n') \in E^*(t)$ en $\exists n'' : (n', n'') \in S^+(t)$ en $n'' \in N_{kitchen}\}$
$\quad \Updownarrow$ invullen
in $N_{kitchen}$ zit enkel knoop 9, daardoor is $n'' = 9$
als $n'' = 9$ dan is $n' \in \{7, 8\}$ (8 is linker broer van 9 en 7 is linker broer van 8)
als $n' = 7$ dan is $n \in \{1, 3\}$ (3 is ouder van 7 en 1 is voorouder)
ook als $n' = 8$ is $n \in \{1, 3\}$ (3 is ouder van 8 en 1 is voorouder)
dan zijn de mogelijke koppels (n, n'): {(1,7),(1,8),(3,7),(3,8)}

4.4.3 Minimale XPath expressies en boomautomaten

In het artikel van Schwentick [20] wordt een methode beschreven om automaten te definiëren voor XPath fragmenten. XPath fragmenten zijn eenvoudige XPath expressies met beperkte syntax en worden aangeduid met $XP(L)$ waarbij L een lijst is van de toegelaten componenten. $XP(/, //)$ is bijvoorbeeld het XPath fragment waarbij enkel *child* en *descendant* zijn toegelaten.

Willen we voor een $XP(/, //)$ expressie p een automaat A definiëren, doen we dit als:

- A doorloopt t van de wortel naar de bladeren en selecteert hierbij niet-deterministisch een pad.
- Op dit pad wordt voor elke knoop v de verste positie in p berekend die overeenkomt met het pad van de wortel tot v. Deze positie wordt als toestand aan de knoop gegeven.
- Knopen die niet op het pad liggen, krijgen als toestand s_*.
- s_* en de toestand van de laatste positie in p worden als eindtoestand gemarkeerd.
- De automaat accepteert een boom t als er een run is waarbij in de bladeren van t enkel accepterende toestanden voorkomen.

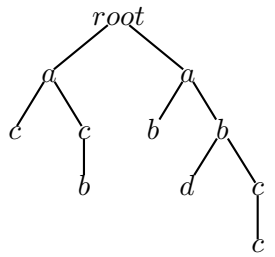
Merk op dat men hier veronderstelt dat de context waarin de expressie uitgevoerd wordt steeds de wortel van de boom is en dat de selectieknoop de laatste knoop in het pad wordt.

Indien de laatste knoop in het pad geen blad van de boom is, zullen alle knopen eronder de toestand s_* krijgen. Deze bomen worden dus ook aanvaard. Ook wordt hier gebruik gemaakt van een top-down beschrijving van de automaat terwijl we in deze tekst steeds boomautomaten bottom-up hebben gedefinieerd. Voor de duidelijkheid zullen we in de voorbeelden de top-down automaat noteren als een bottom-up automaat.

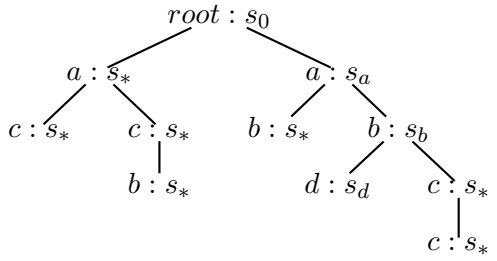
Bijvoorbeeld: voor $p = /a/b//d$ met t zoals in figuur 4.13 zal de automaat $A1$ gedefinieerd worden als:

- $Q_1 = \{s_*, s_0, s_a, s_b, s_d\}$ Merk op dat het gebruik van deze eenvoudige namen voor de toestanden enkel mogelijk is omdat er geen enkel symbool dubbel gebruikt wordt in de expressie. Voor andere expressies kan s_x worden gebruikt met x de positie in de expressie waar we ons bevinden.
- $\Sigma = \{a, b, c, d\}$
- $F_1 = s_0$ Indien de wortel toestand s_0 heeft wordt de boom aanvaard
- De transitiefunctie δ als:
  - $\delta_1(s_0, root) = s_*^* s_a s_*^*$ Dit wil zeggen dat een knoop enkel het label s_0 krijgt als één van de kinderen het label s_a heeft en de andere allemaal s_*.
  - $\delta_1(s_a, a) = s_*^* s_b s_*^*$
  - $\delta_1(s_b, b) = s_*^* s_d s_*^*$
  - $\delta_1(s_d, d) = s_*^* \mid s_*^* s_d s_*^*$
  - $\delta_1(s_d, x) = s_*^* s_d s_*^*$ voor alle $x \in \Sigma$ en $x \neq d$
  - $\delta_1(s_*, x) = s_*^*$ voor alle $x \in \Sigma$

Deze automaat zal alle bomen aanvaarden waarvan er een run bestaat die de wortel het label s_0 geeft. Figuur 4.14 toont een accepterende run op t.



Figuur 4.13: De boom t



Figuur 4.14: Een geldige run op de boom t

4.4.4 Core XPath en boomautomaten

In [13] wordt bewezen (Stelling 1.39) dat elke Core XPath expressie gesimuleerd kan worden door middel van een boomautomaat. In het bewijs wordt er vertrokken van eenvoudige automaten voor simpele XPath expressies die gecombineerd worden tot automaten voor complexere expressies. Voor deze combinaties worden er constructies op boomautomaten beschreven zoals substitutie, unie, doorsnede, verschil en complement. Ook wordt er aangetoond dat de klasse van de reguliere boomtalen gesloten is onder deze constructies.

Het uitschrijven van automaten voor Core XPath expressies kan heel wat tijd vergen. Als voorbeeld starten we met een automaat voor de expressie $\Rightarrow^+ :: a :$ (merk op dat de context en selectieknoop bij deze expressie steeds gelijk zijn)

Beschouw de (bottom-up) boomautomaat $A1 = (Q, \Sigma, \delta, F)$ met als toestanden q_a voor knopen met het label a, q_-^- voor de context/selectie knoop en q_* voor niet-relevante knopen. Laat vervolgens ook q_{true} de toestand worden van alle knopen n waarvan de toestanden van de kinderen voldoen aan $q^* q_-^- q^* q_a q^*$ met $q \in Q$. Dit wil zeggen dat tussen de kinderen van n zowel de contextknoop als een knoop met label a zitten, en de a-knoop zit meer naar rechts. Dan aanvaardt deze automaat bomen die voldoen aan de expressie $\Rightarrow^+ :: a$ als er een knoop gelabeld is met q_{true}.

Op dezelfde manier definiëren we een automaat $A2$ die $\Downarrow^* :: *$ simuleert:

- $\delta(q_{true}, x) = y$ als x een overlijnde knoop is en y een onderlijnde knoop of als $x = y$
- q_* anders
- Deze automaat aanvaardt een boom als er een knoop is met het label q_{true}

Vervolgens kijken we naar de expressie $\Downarrow^* :: [\Rightarrow^+ :: a]$:
De automaat A die deze expressie simuleert, construeren we zo dat: $\mathcal{L}(A) = \mathcal{L}(A2) \cap \mathcal{L}(A1')$. Waarbij $A1'$ de automaat is die bomen aanvaardt waarin de context/selectie knoop van $A1$ een selectieknoop is en één van de andere knopen gemarkeerd wordt als contextknoop. Voor meer uitleg verwijs ik naar [13] bij het bewijs van Stelling 1.39.

4.4.5 Equivalentie van XPath expressies

Om inclusie van XPath expressies aan te tonen, kunnen we zeggen dat een expressie κ_1 een deel is van expressie κ_2 als er geen enkel element is van κ_1 dat niet in κ_2 zit. Dit is ook de methode die Schwentick gebruikt in zijn artikel [20] voor XPath expressies in aanwezigheid van een DTD: eerst construeren we twee boomautomaten, één voor κ_1 en één voor κ_2. Dit is steeds mogelijk (zie sectie 4.4.3 en [13]). Dan nemen we het complement van de tweede automaat, dit geeft een automaat voor alle bomen die niet aan κ_2 voldoen ($\neg\kappa_2$). Vervolgens "plakken" we deze automaten achter elkaar met de product constructie en bekomen we een automaat dat alle tegenvoorbeelden van de inclusie aanvaardt. Indien de uiteindelijke automaat dan een niet-lege set bomen aanvaardt die voldoen aan de DTD, is $\kappa_1 \not\subseteq \kappa_2$. Om na te kijken of de bomen aan een DTD voldoen, kunnen we zoals in sectie 4.3.4 ook een automaat definiëren. We kunnen dan de reeds gevonden automaat uitbreiden tot een automaat voor κ_1, $\neg\kappa_2$ en de DTD. Dit doen we door het product van de drie automaten te berekenen. Om te kijken of een taal, aanvaard door een boomautomaat, al dan niet leeg is, wordt de lezer doorverwezen naar [13].

Determiniseren

Om het complement van een automaat $B = (Q, \delta, F)$ te berekenen, moet er eerst voor gezorgd worden dat B een deterministische automaat is.

Het determiniseren van een automaat wordt uitgelegd in [13] bij de constructies op boomautomaten en werkt als volgt. Zij $B_1 = (Q_1, \delta_1, F_1)$ een niet-deterministische boomautomaat. We construeren een deterministische boomautomaat $B = (Q, \delta, F)$ die dezelfde taal aanvaardt als A_1.

- Q is de verzameling van alle mogelijke deelverzamelingen van Q_1
- F bevat alle deelverzamelingen van Q_1 die een element van F_1 bevatten
- voor elke toestand $R \in Q$, elk woord $R_1..R_k \in Q^*$ en elk symbool $\sigma \in \Sigma$ definiëren we de transitie $\delta(R, \sigma)$ zodanig dat:

$$\begin{gathered} R_1..R_k \in \delta(R, \sigma) \\ \Updownarrow \\ R = \{q \in Q_1 \mid \exists q_1 \in R_1, .., q_k \in R_k : q_1..q_k \in \delta(q, \sigma)\} \end{gathered}$$

  Maken we nu gebruik van de substitutie $f(q) = \{S \mid S \subseteq Q_1 \text{ en } q \in S\}$, dan krijgen we:

$$\delta(R, \sigma) = \bigcap_{q \in R} f(\delta_1(q, \sigma)) - \bigcup_{q \in Q_1 - R} f(\delta_1(q, \sigma))$$

We moeten voor een gegeven $\sigma \in \Sigma$ op elke transitie $\delta_1(q, \sigma)$ in de automaat de substitutie toepassen. Dan nemen we de doorsnede (respectief unie) van alle substituties van transities met $q \in R$ (respectief ($q \in Q_1 - R$). Het verschil van deze twee verzamelingen (doorsnede - unie) is de nieuwe transitie $\delta(R, \sigma)$.

Aangezien we werken met boomautomaten kunnen we zeggen dat voor elke $q \in Q_1$ en $x \in \Sigma$ geldt dat $\delta(q, x)$ een reguliere taal is over Q_1. Reguliere talen zijn gesloten onder unie, doorsnede, verschil en substituties. Bijgevolg is $\delta(R, \sigma)$ ook een reguliere taal.

Het determiniseren van een automaat heeft wel als nadeel dat de nieuwe automaat exponentieel groter is dan de originele automaat.

Uitrekenen van een substitutie

De substituties in de berekening van de deterministische automaat zijn, in het geval van boomautomaten, steeds substituties op reguliere talen. We bekijken een voorbeeld om een beter beeld te hebben op de resultaten. Dit uitgewerkte voorbeeld staat verder in dit hoofdstuk.

Bekijk de substitutie $f(q) = \{S \mid S \subseteq Q_1 \text{ en } q \in S\}$ voor de voorbeeldtransitie $\delta(q, x) = q_0^* q_1 q_0^*$ van een boomautomaat waarbij de reguliere taal wordt voorgesteld als een reguliere expressie E over toestanden.

$$f(\delta(q, x)) = f(q_0^* q_1 q_0^*) = f(E)$$

Als we deze transitie willen uitrekenen, komen we al snel uit aan een oneindige berekening.

$$f(q_0^* q_1 q_0^*) = \{f(q_1), f(q_0 q_1), f(q_0 q_1 q_0), f(q_0 q_0 q_1), ...\}$$

Om dit op te lossen, bekijken we de automaat B_1 die de reguliere taal E herkent en herschrijven we die automaat tot een automaat die de taal $f(E)$ aanvaard. Merk op dat q_0 en q_1 nu symbolen uit het alfabet zijn, daarom nemen we cijfers als toestanden.

B_1:

- $Q_1 = \{0, 1\}$ met 0 als begintoestand en 1 als eindtoestand
- We nemen als alfabet $\Sigma_1 = \{q_0, q_1\}$
- $\delta_1(0, q_0) = 0$
- $\delta_1(1, q_0) = 1$
- $\delta_1(0, q_1) = 1$

De automaat B_2 die $f(\mathcal{L}(B1))$ aanvaardt, wordt dan:

- $Q_2 = \{0, 1\}$ met 0 als begintoestand en 1 als eindtoestand
  De verzameling toestanden, begin- en eindtoestanden blijven gelijk
- het alfabet $\Sigma_2 = \{\emptyset, \{q_0\}, \{q_1\}, \{q_0, q_1\}\}$ bevat alle deelverzamelingen van Σ_1
- Voor elke transitie $\delta(t, \sigma) = t'$ in B_1 en elke $\sigma' \in f(\sigma)$ voegen we een transitie $\delta_2(t, \sigma') = t'$ toe:
  - $\delta_2(0, \{q_0\}) = 0$
  - $\delta_2(0, \{q_0, q_1\}) = 0$
  - $\delta_2(1, \{q_0\}) = 1$
  - $\delta_2(1, \{q_0, q_1\}) = 1$
  - $\delta_2(0, \{q_1\}) = 1$
  - $\delta_2(0, \{q_0, q_1\}) = 1$

Bestuderen we deze automaat, merken we dat deze de volgende taal aanvaardt:
$(\{q_0\} \mid \{q_0, q_1\})^*$ $(\{q_1\} \mid \{q_0, q_1\})$ $(\{q_0\} \mid \{q_0, q_1\})^*$. Hierin herkennen we de oorspronkelijke vorm van de reguliere expressie, waarbij elk symbool q vervangen is door de conjunctie van de elementen van de substitutie $f(q) = \{S \mid S \subseteq Q_1 \text{ en } q \in S\}$ van q. Dit noteren we verder als $f(q_0)^* f(q_1) f(q_0)^*$.
Hieruit besluiten we dat voor een reguliere expressie $r = r_1 ... r_n$ de substitutie gelijk is aan: $f(r) = f(r_1) ... f(r_n)$.

Voorbeeld voor equivalentie van XPath expressies

Zij $p = /a/b//d$ en $q = //c$ met de boom t als in figuur 4.13. In het voorbeeld uit 4.4.3 hebben we reeds een eenvoudige automaat geconstrueerd voor de XPath expressie $p = /a/b//d$, we hebben deze $A1$ genoemd met $Q1$ de verzameling toestanden en $F1$ de accepterende toestanden. Voor q kunnen we een gelijkaardige automaat $A2$ maken als:

- $Q2 = \{q_*, q_0, q_c\}$ We gebruiken andere toestanden als bij $A1$ om straks de productconstructie te verduidelijken
- $\Sigma = \{a, b, c, d\}$
- $F2 = \{q_0\}$
  - $\delta_2(q_0, root) = q_*^* q_c q_*^*$
  - $\delta_2(q_c, c) = q_*^* \mid q_*^* q_c q_*^*$
  - $\delta_2(q_c, x) = q_*^* q_c q_*^*$ voor alle $x \in \Sigma$ met $x \neq c$ en $x \neq root$
  - $\delta_2(q_*, x) = q_*^*$ voor alle $x \in \Sigma$

Nu gaan we $A2$ determiniseren tot een automaat $A2'$ als:

- $Q2' = \{\emptyset, \{q_0\}, \{q_c\}, \{q_*\}, \{q_0, q_c\}, \{q_c, q_*\}, \{q_0, q_*\}, \{q_0, q_c, q_*\}\}$
  Alle mogelijke deelverzamelingen van $Q2$
- $F2' = \{\{q_0\}, \{q_0, q_c\}, \{q_0, q_*\}, \{q_0, q_c, q_*\}\}$
  Alle deelverzamelingen van $Q2$ die elementen van $F2$ bevatten
- Vervolgens berekenen we $\delta(R, x)$ voor elke $R \in Q$ en $x \in \Sigma$ als:
  $\delta'_2(R, x) = \bigcap_{q \in R} f(\delta_2(q, x)) - \bigcup_{q \in Q_2 - R} f(\delta_2(q, x))$
  - $\delta'_2(\{q_0\}, root) = f(\delta_2(q_0, root)) - (f(\delta_2(q_*, root)) \cup f(\delta_2(q_c, root)))$
    $= f(q_*)^* f(q_c) f(q_*)^* - (f(q_*)^* \cup \emptyset)$
    $= f(q_*)^* (\{q_c\} \mid \{q_c, q_0\}) f(q_*)^*$
  - $\delta'_2(\{q_*\}, root) = (\{q_*\} \mid \{q_*, q_0\})^*$
  - $\delta'_2(\{q_*\}, c) = f(q_*)^* (\{q_*\} \mid \{q_*, q_0\}) f(q_*)^*$
  - $\delta'_2(\{q_*\}, x) = (\{q_*\} \mid \{q_*, q_0\})^* \ \forall x \in \Sigma, x \neq root, x \neq c$
  - $\delta'_2(\{q_c\}, c) = f(q_*)^* (\{q_c\} \mid \{q_c, q_0\}) f(q_*)^*$
  - $\delta'_2(\{q_c\}, x) = f(q_*)^* (\{q_c\} \mid \{q_c, q_0\}) f(q_*)^* \ \forall x \in \Sigma, x \neq root, x \neq c$
  - $\delta'_2(\{q_0, q_*\}, root) = f(\delta_2(q_0, root)) \cap f(\delta_2(q_*, root)) - f(\delta_2(q_c, root))$
    $= f(q_*)^* f(q_c) f(q_*)^* \cap f(q_*)^* - \emptyset$
    $= f(q_*)^* (\{q_*, q_c\} \mid \{q_*, q_c, q_0\}) f(q_*)^*$
  - $\delta'_2(\{q_c, q_*\}, c) = f(q_*)^* (\{q_*, q_c\} \mid \{q_*, q_c, q_0\}) f(q_*)^*$
  - $\delta'_2(\{q_c, q_*\}, x) = f(q_*)^* (\{q_*, q_c\} \mid \{q_*, q_c, q_0\}) f(q_*)^* \ \forall x \in \Sigma, x \neq root, x \neq c$

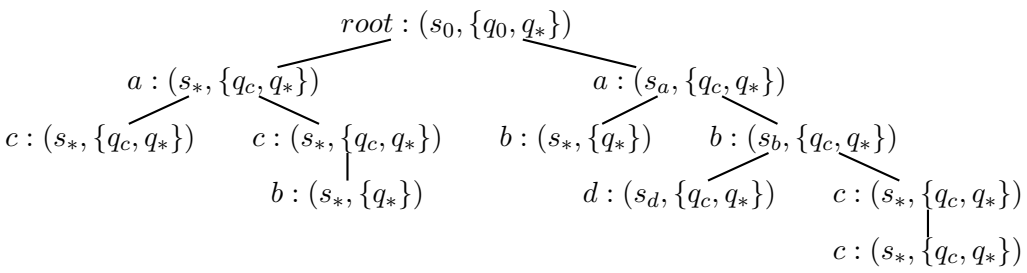
Nu hebben we een deterministische automaat $A2'$ waarvan we het complement kunnen nemen. De automaat $A2''$ heeft dezelfde verzameling toestanden en dezelfde transities als $A2'$.

De verzameling eindtoestanden wordt echter $Q2' - F2'$:

- $F2'' = \{\emptyset, \{q_c\}, \{q_*\}, \{q_c, q_*\}\}$

Nu we eindelijk een automaat hebben voor $\neg q$ kunnen we terug naar het probleem van de inclusie. We beschouwen de automaten $A1$ en $A2''$. Vervolgens combineren we deze twee automaten tot één automaat met behulp van constructie uit [19]. De nieuwe automaat wordt dan:

- $Q = Q1 \times Q2''$ De toestanden worden koppels (s, q) van toestanden.
- $\Sigma = \{a, b, c, d\}$
- $F = F1 \times F2''$
- $\delta((s, q), x) = (s_1, q_1)...(s_n, q_n)$ met $s_1...s_n$ (respectief $q1...q_n$) de toestanden die de kinderen van een knoop met label x krijgen van $A1$ (respectief $A2''$) als de ouder het label s (respectief q) heeft. Voor elke knoop in een boom krijgen we dus een koppel (s, q) toestanden waarbij s de toestand is verkregen van $A1$ en q de toestand van $A2$.



Figuur 4.15: Een run op de boom t

Deze automaat aanvaardt vervolgens alle bomen waarvan de bladeren toestanden hebben in F. Als we onze boom t uit 4.13 testen met behulp van deze automaat, krijgen we als mogelijke run (aangezien $A1$ niet-deterministisch is) de run uit figuur 4.15.
We hebben nu een automaat geconstrueerd die alle bomen aanvaardt, die wel aan p maar niet aan q voldoen. Op gelijkaardige wijze kunnen we deze automaat vermenigvuldigen met een automaat voor een DTD. Op die manier bekomen we een automaat die alle tegenvoorbeelden aanvaardt die voldoen aan de DTD.

4.5 Besluit

In de cursus theoretische informatica voor het tweede bachelorjaar informatica is er geen ruimte om boomautomaten op te nemen zonder andere leerstof weg te laten. De docent heeft de keuze deze leerstof aan te bieden als keuzeonderwerp of in een keuzevak. Verder is het belangrijk dat er overleg is tussen de docenten van verschillende vakken zodat onderlinge integratie en geleidelijkheid gegarandeerd kunnen worden. Hierbij denken we in de eerste plaats aan de cursus "Technologie van Multimediasystemen en -software" [**?**, XMLcursus]aar de student kennismaakt met XML en XPath.

In dit hoofdstuk hebben we eerst boomautomaten gedefinieerd. Vervolgens hebben we gekeken op welke manier boomautomaten kunnen helpen bij het gebruik van XML, DTD's en XPath. We hebben aangetoond dat we boomautomaten kunnen gebruiken voor het valideren van XML bomen waarbij we de DTD omzetten in een boomautomaat. Hierbij hebben we, voor het ondubbelzinnig definiëren van DTD's, ook gebruik gemaakt van one-unambiguous reguliere expressies. We testen of een reguliere expressie one-unambiguous is door gebruik te maken van de Glushkov-constructie van hoofdstuk 3. Als laatste hebben we boomautomaten voor XPath expressies gedefinieerd en hebben we deze boomautomaten gebruikt om te kijken of twee XPath expressies equivalent zijn.

Hoofdstuk 5

Stringautomaten en pattern matching

5.1 Didactische wenken

In de cursus algoritmen en datastructuren van het tweede bachelorjaar informatica staan verschillende zoekalgoritmen. Deze algoritmen zoeken de positie van een karakter in een string. Een uitbreiding op deze algoritmen zijn zoekalgoritmen voor een reeks karakters (een patroon) in een string of tekst. Deze algoritmen worden pattern matching algoritmen genoemd.

Er bestaan een aantal pattern matching algoritmen op basis van stringautomaten. Dit is bijgevolg een aangewezen onderwerp om de leerstof uit de cursus algoritmen en datastructuren te integreren met theoretische informatica. Het is de keuze van de docenten in welk (keuze)vak deze leerstof gegeven wordt. De docent kan de studenten actief bezighouden door deze algoritmen als zelfstudie aan te bieden.

Aangezien er ook pattern matching algoritmen bestaan die niet gebaseerd zijn op de automatentheorie kunnen we gaan differentiëren. De algoritmen in dit hoofdstuk bieden we aan aan studenten die gekozen hebben voor meer theoretische informatica. De andere studenten krijgen pattern matching algoritmen zonder automaten.

5.2 Wat is pattern matching

Tekstverwerkingsprogramma's bieden de gebruikers de mogelijkheid op zoek te gaan naar een bepaald woord in de tekst, virusscanners zoeken de "handtekening" van een virus in de (binaire) code van een bestand, zoekmachines op het internet geven alle pagina's weer waarin een bepaalde tekst voorkomt en zelfs databases waarin DNA strings worden opgeslagen bieden de mogelijkheid te zoeken naar een bepaalde DNA string.

Al deze voorbeelden zijn toepassingen van hetzelfde principe: pattern matching. Pattern matching of string matching is het proces om te bepalen of en waar een bepaalde string (het patroon) voorkomt in een andere string (een tekst). Er bestaan verschillende algoritmen om dit te doen: de meest bekende zijn het naïeve algoritme, Knuth-Morris-Pratt (KMP) en Boyer-Moore (BM). We kunnen deze zoekalgoritmen indelen op basis van de benodigde

geheugenruimte, de snelheid waarmee ze werken of het aantal noodzakelijke karaktervergelijkingen om de locaties te vinden waar een patroon voorkomt. Ook zijn er algoritmen die in een preprocessing stap bepaalde gegevens uit het patroon dan wel uit de tekst halen. En sommige algoritmen vergelijken het patroon van links naar rechts en andere van rechts naar links.

In dit hoofdstuk geven we een overzicht van KMP, BM en enkele interessante pattern matching algoritmen die gebruik maken van automaten. Daarbij maken we de volgende afspraken:

- Het patroon $x = x_0...x_{m-1}$ en de tekst $y = y_0...y_{n-1}$ zijn strings in het Latijnse alfabet of een deelverzameling ervan, het patroon heeft lengte m en de tekst heeft lengte n
- Het alfabet heeft σ symbolen
- We zoeken op welke positie(s) x voorkomt in y, de bedoeling is dus alle voorkomens van x in y terug te vinden en niet enkel het eerste.
- Voorbeelden gaan uit van $x = baabb$ en $y = babaabaabb$ met het alfabet $\{a, b, c\}$

De beschrijving van de algoritmen en hun complexiteit zijn terug te vinden in [28], voor het KMP algoritme is ook [27] gebruikt.

5.3 Overzicht van algoritmen

5.3.1 Naïeve oplossingsmethode

Een zeer voor de hand liggende, doch naïeve methode om een patroon te zoeken in een tekst is het aflopen van alle verschillende mogelijke posities van het patroon in de tekst. Zoeken we bijvoorbeeld het patroon *baabb* in de tekst *babaabaabb*, kunnen we de werkwijze uitschrijven zoals in figuur 5.1.

1. baabb babaabaabb	7. **ba**abb ba**ba**abaabb	13. **b**aabb babaa**b**aabb
2. **b**aabb **b**abaabaabb	8. **baa**bb ba**baa**baabb	14. **ba**abb babaa**ba**abb
3. **ba**abb **ba**baabaabb	9. **baab**b ba**baab**aabb	15. **baa**bb babaa**baa**bb
4. **baa**bb **bab**aabaabb	10. **baabb** ba**baaba**aabb	16. **baab**b babaa**baab**b
5. **b**aabb b**a**baabaabb	11. **b**aabb bab**a**abaabb	17. **baabb** babaa**baabb**
6. **b**aabb ba**b**aabaabb	12. **b**aabb baba**a**baabb	

Figuur 5.1: Naïef zoeken van "baabb" in "babaabaabb"

Voor elke letter van de tekst wordt er gekeken of deze overeenkomt met de eerste letter van het patroon. Indien dit het geval is, vergelijken we de volgende letter van de tekst en het patroon tot het einde van het patroon bereikt is (match gevonden) of er een mismatch optreedt.

Complexiteit

Dit algoritme heeft tijdscomplexiteit $O(mn)$ en er worden $2n$ karaktervergelijkingen verwacht.

5.3.2 Knuth Morris Pratt algoritme (KMP)

Het naïeve algoritme doet vaak onnodige vergelijkingen. Zo weten we bijvoorbeeld in stap 10 in figuur 5.1 al dat $y[3]$ en $y[4]$ a zijn, en is het dus nutteloos deze nog te vergelijken met b (de eerste letter van ons patroon) zoals in stap 11 en 12. We zouden eigenlijk onmiddellijk kunnen overschakelen naar stap 13 waar $y[5]$ gematcht wordt. Dit is net wat het algoritme van Knuth-Morris-Pratt (KMP) doet. Hiervoor wordt er eerst gekeken op welke manier het patroon overlapt met een deelpatroon.

Beschouw bijvoorbeeld dezelfde stap uit het naïeve algoritme:

10. **baabb**
    ba**baaba**abb

Op dit moment zijn we een deelpatroon *baab* tegengekomen in de tekst.
Nu gaan we bekijken op welke manier het patroon *baabb* kan overlappen met dit deelpatroon.

```
verschuiving = 1   verschuiving = 2   verschuiving = 3
  b a a b            b a a b            b a a b
    b a a b b          b a a b b          b a a b b
```

De maximale overlapping in dit voorbeeld is van lengte 1, enkel de suffix b van het deelpatroon *baab* is een prefix van het patroon *baabb*. Dit wil zeggen dat we pas drie (lengte deelpatroon min lengte overlapping) posities verder een mogelijk voorkomen van het patroon kunnen vinden, we zullen dus drie iteraties verder springen.

Om de overlap te berekenen hebben we enkel het patroon p en het deelpatroon nodig. Het deelpatroon is volledig bepaald door zijn lengte. Voor alle mogelijke posities kan de overlap volledig vóór het pattern matching algoritme berekend worden. Als vereenvoudiging kunnen we een tabel O definiëren met $O[j]$ de lengte van overlapping van het patroon en het deelpatroon $P[0..j]$.

Voor *baabb* bekomen we dan de volgende tabel:

i	0	1	2	3	4	5
x[i]	b	a	a	b	b	
O[i]	-1	0	0	-1	1	1

Nu kunnen we nog een optimalisatie toepassen op het algoritme. Als we een overlapping tegenkomen en we slaan hierdoor een (aantal) iteraties over, beginnen we in de volgende 'ronde' met het vergelijken van de karakters waarvan we reeds weten dat ze overeenkomen aangezien ze net de overlap bepalen. Bijgevolg mogen we net zoveel karakters overslaan als de lengte van de overlap.

Complexiteit

Dit algoritme heeft nog steeds een geneste herhalingsstructuur, dus kunnen we ons afvragen of het wel een optimalisering is ten opzichte van het naïeve algoritme. Een nadere bestudering [29] wijst echter uit dat de tijdscomplexiteit van het KMP algoritme $O(n)$ is met maximaal $2n$ vergelijkingen. Er moet wel nog rekening gehouden worden met de berekening van de overlap tabel, deze berekening gebeurt in $O(m)$ tijdscomplexiteit. Hierdoor is de totale tijdscomplexiteit in $O(m+n)$.

De tijdscomplexiteit kan berekend worden door te tellen hoe vaak we de vergelijking $y[i+j] == x[j]$ uitvoeren. Deze vergelijkingen splitsen we op in twee groepen: de vergelijkingen die waar teruggeven (positieve vergelijkingen) en de vergelijkingen die onwaar teruggeven (negatieve). Als een vergelijking positief is, kennen we de waarde van $y[i+j]$. In volgende iteraties zullen we dan over deze $y[i+j]$ heen springen als er een overlap gevonden werd of zal de vergelijking $y[i+j] == x[j]$ onwaar teruggeven. Voor elke mogelijke positie van $y[]$ zal er dus slechts een positieve vergelijking voorkomen, bijgevolg komen er slechts n positieve vergelijkingen voor.

Aan de andere kant kan er in de binnenste lus slechts éénmaal een negatieve vergelijking voorkomen aangezien dit de lus afbreekt. In totaal kunnen er dus slechts n (eigenlijk $n-m$) negatieve vergelijkingen voorkomen. Het gehele algoritme maakt dus maximaal $2n$ vergelijkingen en werkt in tijd $O(n)$, een duidelijke verbetering.

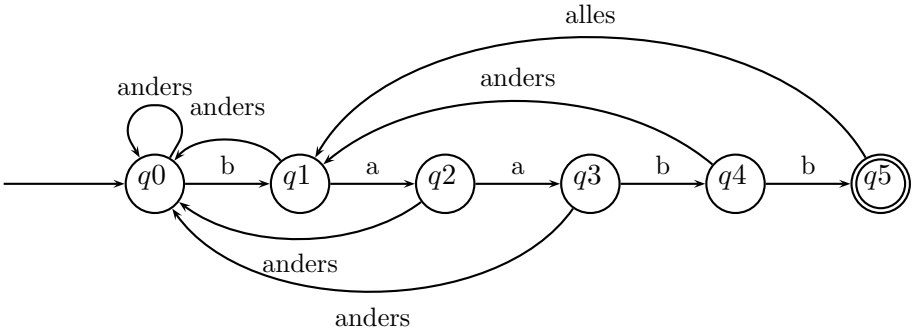
KMP en automatentheorie

Als we het voorgaande algoritme bekijken, kunnen we er zeer eenvoudig een automaat van maken [29]. Een starttoestand q_0, toestanden $q_1 ... q_m$ met m = de lengte van het patroon en de overgangen zijn bepaald als:

- Vanuit toestand q_i gaan we naar toestand q_{i+1} als we de letter gelezen hebben die overeenkomt met de i-de letter van het patroon
- We gaan terug naar een vorige toestand als er een andere letter gelezen werd, de toestand waar we terecht komen, komt overeen met de overlap en we lezen de laatst gelezen letter opnieuw in.

Deze automaat accepteert een string als de laatste toestand (q_m) bereikt wordt. De visualisatie in figuur 5.2 maakt alles duidelijker.

Bij het lezen van een verkeerd karakter wordt er telkens teruggegaan naar de eerste toestand tenzij er overlap was. In dit voorbeeld is er overlap bij het bereiken van toestand q_4 en wordt er teruggegaan naar toestand q_1 omdat er slechts een overlap van één karakter is.



Figuur 5.2: KMP Automaat voor het patroon baabb

Merk op dat er ook overlap is bij toestand q_5. Hoewel het hele patroon reeds gevonden is, kunnen we door de pijl van q_5 naar q_1 zorgen dat alle voorkomens van het patroon gezocht worden, zelfs als ze elkaar overlappen. We benadrukken wel dat het zeer ongewoon is dat bepaalde karakters opnieuw ingelezen moeten worden als we terugkeren naar een vorige toestand.

5.3.3 Het Boyer-Moore algoritme (BM)

In tegenstelling tot het KMP algoritme vergelijkt het BM algoritme eerst het laatste symbool van het patroon om vervolgens het voorlaatste symbool te vergelijken, het patroon wordt dus van links naar rechts langs de tekst geschoven en binnen het patroon worden de symbolen van rechts naar links vergeleken met de tekst. Indien we een verkeerd symbool tegenkomen, schuiven we het patroon door naar rechts. Het BM algoritme wordt beschouwd als het efficiëntste algoritme voor de meeste toepassingen.

Beschouwen we de situatie waarbij een verkeerd symbool gevonden wordt op $x[i]$, we weten dan dat $y[i+j] \neq x[i]$ maar alle volgende symbolen wel matchen ($x[i+1...m-1] = y[i+j+1...j+m-1]$). Noem deze string u, u is een suffix van x. Dan wordt in het BM algoritme het patroon verschoven met het maximum van de twee shift functies good-suffix-shift en bad character shift.

De good-suffix-shift verschuift het patroon zodanig dat

- de u die we vonden in y overeenkomt met het meest rechtse voorkomen van u in x dat wordt voorafgegaan door een ander symbool als $x[i]$
- de langste suffix van u gelijk ligt met een prefix van x

De bad-character-shift:

- zal het karakter $y[i+j]$ uitlijnen met het meest rechtse voorkomen van dat karakter in x.
- indien $y[i+j]$ niet meer voorkomt in x wordt het eerste karakter van x uitgelijnd met $y[i+j+1]$.

Voor ons voorbeeld geeft dit de volgende tabellen:

Good-suffix-shift:

i	0	1	2	3	4
	b	a	a	b	b
gs[i]	4	4	4	1	2

Bad-character-shift: (telt van rechts naar links maar slaat het laatste symbool over)

c	a	b	c
bc[c]	2	1	5

Zoeken we dan in de tekst *babaabaabb* dan bekomen we de volgende stappen:

1. *babaabaabb*
   baabb
   mismatch → shift by $2 = max(gs[i], bc[a](m-1)+i)$ met $i = 4$
2. *babaabaabb*
   baabb
   mismatch → shift by $2 = max(gs[i], bc[a](m-1)+i)$ met $i = 4$
3. *babaabaabb*
   baabb
   match → we kijken naar het volgende symbool
4. *babaabaabb*
   baabb
   mismatch → shift by $1 = max(gs[i], bc[a](m-1)+i)$ met $i = 3$
5. *babaabaabb*
   baabb
   matchen tot we mismatch tegenkomen of het patroon herkend is
6. *babaaBAABB*
   BAABB
   het patroon is gevonden shift by $4 = gs[0]$

Complexiteit:

De tabel good-suffix-shift wordt in m berekeningen opgesteld, één voor elke letter van het patroon. De bad-character-shift tabel heeft een waarde nodig voor elke letter van het alfabet. Als we stellen dat het alfabet uit σ karakters bestaat kunnen we zeggen dat deze tabel in $O(\sigma)$ tijd berekend wordt. De hele preprocessing stap gebeurt dan in tijd $O(m + \sigma)$.

Het zoeken zelf gebeurt in $O(mn)$ maar er worden slechts $3n$ karaktervergelijkingen verwacht indien we zoeken naar een niet-periodisch patroon. Het snelste dat dit algoritme loopt, is $O(n/m)$. Dit is het absolute minimum dat een algoritme nodig heeft indien er enkel preprocessing is van het patroon.

5.3.4 Zoeken met een deterministische eindige automaat

In sectie 5.3.2 hebben we het KMP algoritme gesimuleerd met een automaat. Er zijn echter nog andere manieren om een automaat op te stellen zodat deze een patroon in een tekst herkent.

Beschouwen we de volgende automaat:

- de verzameling toestanden $Q = q_0 q_m$, indien we in toestand q_i zijn hebben we de prefix $x[0i-1]$ herkend van het patroon
- q_0 is de begintoestand
- voor alle $0 < i < m-1$ is er een pijl van q_i naar $q_i + 1$ met $x[i+1]$ als label
- verder zijn er terugkerende pijlen van q_i naar q_j met label a (uit het alfabet) als $x[0j-1]$ de langste suffix is van $x[0i-1]a$ die ook een prefix is van x
- q_m is de accepterende toestand

We verduidelijken met een voorbeeld: $x = baabb$, $m = 5$ $Q = q_0, q_1, q_2, q_3, q_4, q_5$ met q_0 als begintoestand en q_5 als accepterende toestand.

De pijl van q_0 naar q_1 heeft label b ($x[0]$), q_1 naar q_2 heeft label b ($x[2]$), en zo verder. Voor elk symbool dat niet in het patroon voorkomt, vertrekt er uit elke toestand een pijl naar q_0, deze tekenen we voor het overzichtelijk te houden niet uit. Nu berekenen we de langste suffix voor elke mogelijke letter:

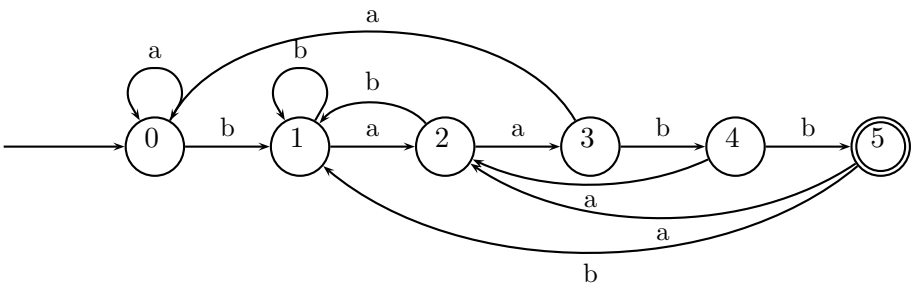
- Indien we in toestand q_1 zitten, hebben we reeds de string b gematcht. Als we een b inlezen kunnen we zeggen dat de suffix b van bb de langste suffix is die ook een prefix is van x, vandaar dat we een lus toevoegen naar q_1.
- In toestand q_2 gaan we verder bij een a en terug naar q_0 bij het lezen van een symbool dat niet in het patroon voorkomt, maar als we een b lezen kunnen we weeral zeggen dat b de langste suffix is van bab die een prefix is van x, we tekenen dus een pijl van q_2 naar q_1.
- Indien we zo verder gaan, bekomen we de automaat zoals in figuur 5.3.

Complexiteit:

De automaat kan opgesteld worden in $O(m + \sigma)$ tijdscomplexiteit. Eénmaal we de automaat hebben, kunnen we in $O(n)$ tijd het woord zoeken in de tekst (indien we veronderstellen dat de automaat is opgeslagen in een direct access table).

Vergelijking met KMP automaat:

In tegenstelling tot het KMP automaat, moet bij deze automaat geen enkel karakter opnieuw gelezen worden. Hierdoor kunnen we zeggen dat elk karakter uit de tekst maximaal 1 keer vergeleken zal worden met een karakter uit het patroon. Dit uit zich in een snellere zoektijd. Aangezien het opstellen van de automaat nu wel meer tijd nodig heeft, is het gebruik van deze automaat enkel nuttig indien we hetzelfde patroon in meerdere teksten moeten zoeken.

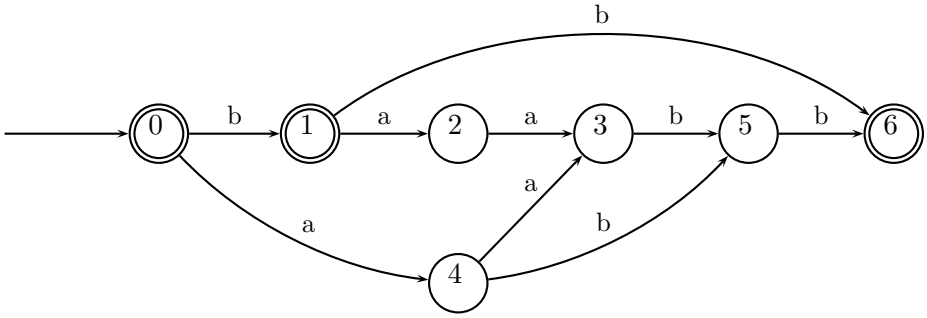


Figuur 5.3: Zoeken met behulp van een deterministische eindige automaat

5.3.5 Forward DAWG Matching algorithm (Suffix automaton)

Dit algoritme maakt gebruik van de zogenaamde kleinste suffix automaat (smallest suffix automaton), de kleinste automaat die alleen die factoren (deelstrings) van een string aanvaardt die puur suffix zijn van die string. Dus de suffix-automaat van het woord w aanvaardt alle strings u waarvoor geldt dat er een string v bestaat over het alfabet zodat $vu = w$.

Voor het voorbeeld *baabb* krijgen we de suffix automaat van figuur 5.4.



Figuur 5.4: Forward DAWG Matching algorithm (Suffix automaton)

In deze automaat komt een toestand q_i soms overeen met verschillende mogelijke paden van q_0 naar q_i maar bemerk dat al deze paden een andere lengte hebben (dit volgt rechtstreeks uit de constructie van de automaat). Op elk pad kunnen we een string lezen in de labels van de pijlen, we kunnen een pad dan ook zien als een string.

Verder worden er in de preprocessing stap twee functies uitgeschreven in een tabel:

- voor elke toestand q in de suffix automaat is $length(q)$ de afstand van het langste pad van q_0 naar q
- voor elke toestand q in de suffix automaat is $S[q]$ de "suffix link" van p, dit is de toestand waarin men terechtkomt als men de langste suffix van het langste pad naar q neemt en vervolgens de karakters van deze suffix inleest vertrekkende bij q_0, waarbij $p \neq q$. Indien $p = q$ nemen we een kortere suffix en volgen weer het pad.

Voor het voorbeeldje *baabb* en de hierboven beschreven suffix automaat krijgen we bijvoorbeeld $S[3] = 4$ want:

- in toestand 3 lezen we het pad *baa* als langste pad
- de langste suffix is *aa* maar als we deze volgen komen we vanuit q_0 terug in toestand 3 uit
- dan bekijken we suffix *a*, hierbij komen we vanuit de startpositie q_0 in toestand 4 deze toestand is dus de suffix link voor toestand 3

De volledige tabellen worden dan:

i	0	1	2	3	4	5	6
S[i]	0	0	4	4	0	1	1
length[i]	0	1	2	3	1	4	5

Het zoeken van een patroon in een tekst gebeurt vervolgens door het sequentieel inlezen van de symbolen van de tekst in de automaat. Indien er geen pijl vertrekt met het juiste label veranderen we de huidige toestand q naar de toestand $S[q]$, we volgen dus de suffix link. Bij het starten en als we geen pijl kunnen volgen, zetten we een teller op 0 die toeneemt telkens we in een nieuwe toestand komen (dus ook als we de suffix link instellen als huidige toestand). Indien we in de eindtoestand[1] komen wordt er naar deze teller gekeken. Is deze gelijk aan de lengte van het patroon, hebben we een volledig voorkomen van dit patroon gevonden.

Het zoeken van *baabb* in *babaabaabb* gaat dan als volgt:

1. $\underline{b}\ a\ b\ a\ a\ b\ a\ a\ b\ b$ teller
   0 0
   We starten in toestand 0 met de teller op 0
2. $b\ \underline{a}\ b\ a\ a\ b\ a\ a\ b\ b$
   0 1 1
   Na het lezen van *b* zitten we in toestand 1 met de teller op 1, nu lezen we *a* en gaan naar toestand 2
3. $b\ a\ \underline{b}\ a\ a\ b\ a\ a\ b\ b$
   0 1 2 2
   vanuit toestand 2 vertrekt geen pijl met label *b* dus wordt de huidige toestand de suffix link van toestand 2: $S[2] = 4$, merk ook op dat de teller is aangepast
4. $b\ a\ \underline{b}\ a\ a\ b\ a\ a\ b\ b$
   4 1
5. $b\ a\ b\ \underline{a}\ a\ b\ a\ a\ b\ b$
   4 5 2
   suffix link: $S[5] = 1$

[1] Er zijn eigenlijk meerdere eindtoestanden, maar enkel één, q_6, waarbij het volledige patroon gelezen kan zijn.

6. $b\ a\ b\ \underline{a}\ a\ b\ a\ a\ b\ b$
   1 1

7. $b\ a\ b\ a\ a\ b\ \underline{a}\ a\ b\ b$
   1 2 3 5 4
   suffix link: $S[5] = 1$

8. $b\ a\ b\ a\ a\ b\ \underline{a}\ a\ b\ b$
   1 1

9. $b\ a\ b\ a\ a\ b\ a\ a\ b\ b$
   1 2 3 5 6 5

We zitten in toestand 6 en de teller is gelijk aan de lengte van het patroon, we hebben dus een match gevonden! De suffix link van 6 is 1, dit is de toestand waarin we verder zouden gaan indien de tekst langer was.

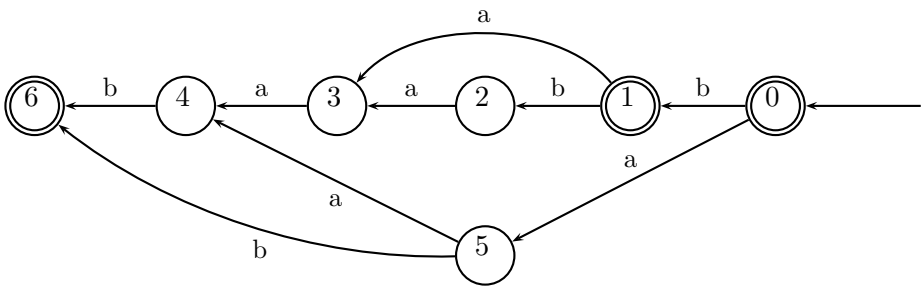
Opmerking:
Indien we in de tekst een symbool tegenkomen dat niet in het patroon gebruikt wordt, beginnen we terug in q_0 en zetten we de teller op 0, we lezen dan gewoon het volgende symbool.

Complexiteit:

Het opstellen van het suffix automaat kan in een tijd lineair met de lengte van het patroon $= O(m)$ en er worden n karaktervergelijkingen uitgevoerd.

5.3.6 Reverse Factor algorithm

Het reverse factor algoritme gebruikt net als het vorige algoritme de kleinste suffix automaat, maar hier wordt de automaat geconstrueerd voor het inverse van het patroon. Dat wil zeggen dat als het patroon *baabb* is we in dit geval de kleinste suffix automaat van bbaab zullen definiëren. Deze automaat ziet er uit als in figuur 5.5.



Figuur 5.5: Reverse Factor algorithm

Voor de rest van het algoritme starten we in toestand q_0 en beschouwen als eerste het symbool op positie $m-1$ en dan verder naar links. We volgen de automaat tot we een karakter tegenkomen waar geen pijl voor voorzien is. Ondertussen houden we een teller bij die weergeeft hoe lang het pad is in de automaat van q_0 tot de laatste eindtoestand die we tegenkwamen. Indien we geen pijl meer kunnen volgen, schuiven we net zoveel door in de tekst als de lengte van het patroon min de waarde van de teller. We zetten de toestand terug op q_0, de teller op 0 en volgen terug de automaat. Indien we op een bepaald moment m gematchte symbolen zijn tegengekomen weten we dat deze het patroon voorstellen. In plaats van de teller op dat moment aan te passen (we zitten immers in een eindtoestand), gaan we de huidige waarde van de teller gebruiken om de shift te berekenen ($m - teller$).

Het zoeken verloopt dan als volgt, we starten in q_0:

1. $\underline{b\ a\ b\ a}\ \mathbf{a}\ b\ a\ a\ b\ b$
   We lezen de letter in die vetgedrukt staat, de toestand waarin we terecht komen noteren we onder de letter en we onderlijnen eindtoestanden. We volgen de automaat tot we geen geldige transitie meer vinden

2. $\underline{b}\ \mathbf{a}\ \underline{b\ a\ a}\ b\ a\ a\ b\ b$
   $\underline{6}\ 4\ 5$
   In toestand 6 vinden we geen pijl met het label a, de eerste eindtoestand die we tegen komen (van rechts naar links) is toestand 6, dit is de derde toestand, bijgevolg gaan we $5-3=2$ (lengte patroon lengte tot aan eindtoestand) posities doorschuiven en starten we terug in toestand 0

3. $b\ a\ \underline{b\ a}\ \mathbf{a}\ \underline{b\ a}\ a\ b\ b$
   $\underline{6}\ 5$
   Na het lezen van a en b vinden we geen transitie voor a in toestand 6, ditmaal is de shift gelijk aan 3

4. $b\ a\ b\ a\ \mathbf{a}\ \underline{b\ a\ a\ b\ b}$
   $\underline{6}\ 4\ 3\ 2\ \underline{1}$
   We hebben vijf karakters kunnen matchen, dit is de langst mogelijke suffix van het patroon namelijk het patroon zelf. De laatste shift komt voorbij het einde van de tekst uit dus we mogen stoppen.

Complexiteit

Het opstellen van de automaat duurt even lang als bij het forward DAWG algoritme ($O(m)$) en het zoeken gebeurt in tijd $O(mn)$.

5.4 Besluit

Met deze algoritmen hebben we vooral willen aantonen dat de toepassingsmogelijkheden van stringautomaten verder gaan als de theoretische informatica. Onderwerpen zoals pattern matching zijn uitermate geschikt voor integratie en differentiatie. Ook is het een ideaal onderwerp om de studenten door middel van zelfstudie actief te laten studeren.

Hoofdstuk 6

Finite state adventures

6.1 Didactische wenken

In de inleiding (Hoofdstuk 1) hebben we reeds vermeld dat studenten vaak niet gemotiveerd zijn voor het vak theoretische informatica. In dit hoofdstuk gebruiken we een creatief voorbeeld om aan de studenten duidelijk te maken wat de Chomsky-hiërarchie is.

Door de inleiding te geven in de vorm van een verhaal kan de docent de aandacht van de leerlingen trekken en tegelijk de opdracht schetsen. Het opsplitsen van de avonturen in levels is enerzijds nodig voor de link te leggen met de Chomsky-hiërarchie maar helpt anderzijds ook voor het geleidelijk verwezenlijken van de onderwijsdoelstellingen.

De docent moet er wel over waken dat alle nodige voorkennis reeds gegeven is. De studenten moeten kennis hebben van eindige automaten, de doorsnede van automaten, pushdown-automaten, petri netten, two-counter machines en de Chomsky-hiërarchie. In de huidige cursus theoretische informatica ([1]) ontbreken echter petri netten, two-counter machines en de Chomsky-hiërarchie. Desondanks blijft dit een interessant voorbeeld om de interesse van de studenten te wekken. De docent kan zich bijvoorbeeld beperken tot de levels 0, 1 en 2.

6.2 Wat zijn finite state adventures?

Finite state adventures of FSA's werden uitgevonden in Duitsland aan de universiteit van München. Wilfried Brauer gebruikte ze in zijn cursus theoretische informatica in de zomer van 2001. Deze cursus was helemaal gericht op bachelor studenten in hun tweede jaar. Een beschrijving vinden we in [30] en [31].

FSA's zijn voorbeelden die gebruikt kunnen worden bij het aanleren van onderwerpen zoals automaten, reguliere en contextvrije talen en Chomsky-hiërarchie. Ze zijn ontworpen om de studenten actief bij de leerstof te betrekken en werken motiverend.

6.3 Methode

6.3.1 Inleiding

Als inleiding tot de leerstof wordt er eerst een verhaaltje verteld over het avontuur dat de docenten en studenten zullen beleven. Het avontuur speelt zich af in de wereld van de grote rivier waar we betoverde deuren, magische bogen en sleutels tegenkomen op de zoektocht naar schatten. Er is echter ook het gevaar van draken die enkel verslagen kunnen worden met een zwaard.

Gelukkig krijgen de avonturiers ook een kaart mee waarop alle gevaren en voorwerpen staan. Deze kaart is een niet-deterministische eindige automaat waarbij de transities symbolen hebben die staan voor alles wat we onderweg kunnen tegenkomen. Deze symbolen kunnen we ook noteren als letters en zijn het alfabet van de automaat. Een voorbeeld van een dergelijke kaart staat in figuur 6.1.

Het alfabet is $\Sigma = \{$

d (dragon / draak),

s (sword / zwaard),

a (arch / boog),

r (river / rivier),

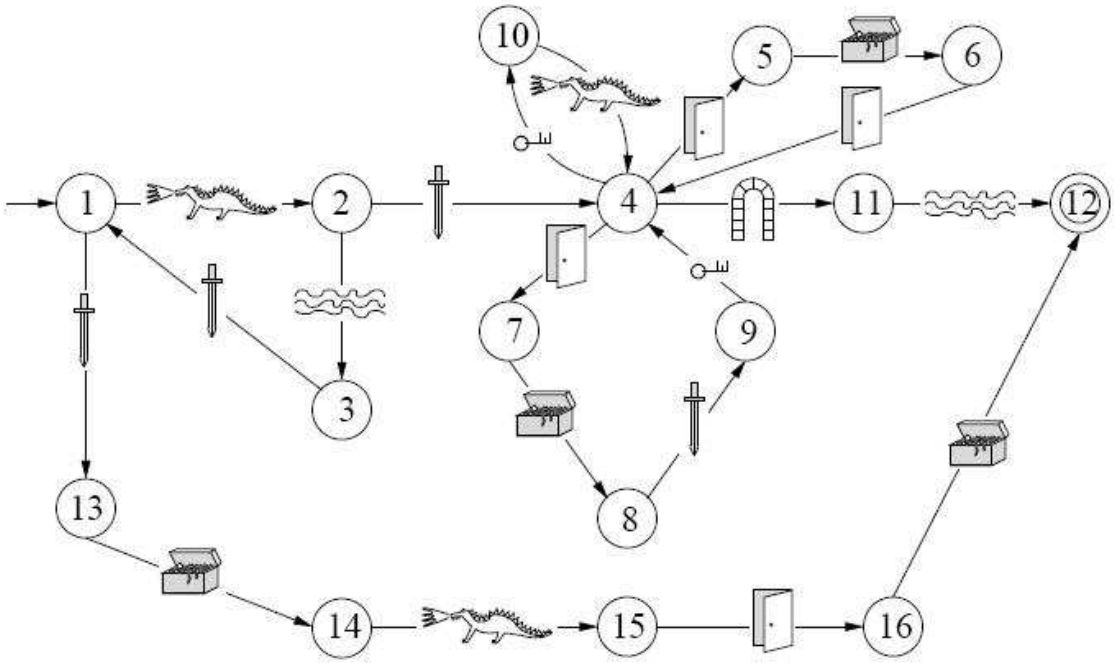
g (gate / deur),

t (treasure / schat),

k (key / sleutel)

$\}$

Als we een pijl volgen waarbij we een voorwerp tegenkomen, noteren we de letter van dit voorwerp. Zo krijgen we bij het wandelen door de wereld een reeks letters die het pad bepalen dat we gevolgd hebben. Om avonturen moeilijker te maken, zullen er eisen gesteld worden aan het pad. Deze eisen bepalen wanneer een pad geldig is en aan welke voorwaarden een bepaalde transitie gevolgd kan worden. Avonturen worden bepaald door een kaart en door de lijst van voorwaarden. Afhankelijk van de moeilijkheidsgraad van de avonturen komen ze terecht in een level tussen 0 en 4.



Figuur 6.1: Een voorbeeld FSA kaart

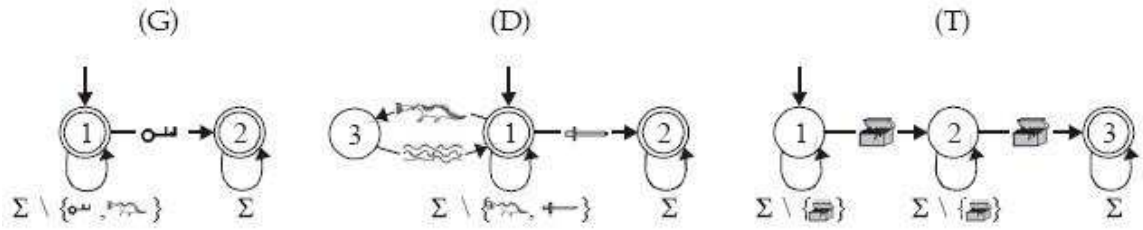
6.3.2 Level 0

In het laagste level moeten de avonturiers een weg vinden die begint op het startpunt van de kaart en eindigt in een eindpunt op de kaart. Er worden verder geen voorwaarden gesteld aan het pad. De avonturier kan dus gewoon door deuren lopen en langs draken. Het avontuur komt met andere woorden overeen met het zoeken van een geldig pad in de eindige automaat M die de kaart voorstelt. De verzameling van geldige paden bevat alle mogelijke wegen die de avonturiers naar de overwinning leiden. Deze verzameling is de taal, aanvaard door de automaat of $L(M)$. We noteren de taal van een avontuur van level 0 als $L(A_0) = L(M)$.

Als we deze avonturen willen plaatsen in de Chomsky-hiërarchie komen we terecht bij de reguliere talen of Chomsky type 3. Dit komt doordat de taal, aanvaard door een eindige automaat, altijd een reguliere taal is. Voor avonturen van level 0 kunnen we in tijd $O(n^2)$ beslissen of het avontuur oplosbaar is, waarbij n gelijk is aan het aantal toestanden in de kaart M. Dit komt namelijk overeen met het zoeken naar de aanwezigheid van een geldig pad in een automaat M met n toestanden (en dus maximaal n^2 pijlen).

6.3.3 Level 1

Nu maken we de avonturen moeilijker. De avonturier kan slechts door een deur indien hij een sleutel in zijn bezit heeft. Eens een sleutel gevonden, kan deze voor alle deuren gebruikt worden. Verder kunnen we slechts voorbij een draak indien we een zwaard hebben gevonden op onze weg. Als alternatief mogen we ook in een rivier springen nadat we door een draak in brand zijn gezet. De laatste eis voor het slagen van het avontuur is het verzamelen van minstens twee schatten.



Figuur 6.2: De automaten voor bogen (G), draken (D) en schatten (S)

Al deze eisen kunnen we voorstellen met behulp van eindige automaten. In figuur 6.2 zien we deze automaten die we G, D en T genoemd hebben. Het avontuur bevat nu enerzijds de kaart-automaat M en anderzijds de drie automaten G, D en T. Om een oplossing te vinden moet een geldig pad (een woord) op de kaart staan (in $L(M)$) en aan de drie andere voorwaarden voldoen (in $L(G)$ en in $L(D)$ en in $L(T)$). De resulterende taal van het avontuur is de doorsnede van de vier talen $L(A_1) = L(M) \cup L(G) \cup L(D) \cup L(T)$.

Als we weten dat de doorsnede van reguliere talen ook regulier is, kunnen we besluiten dat ook avonturen van level 1 van Chomsky type 3 zijn. Verder kunnen we zeggen dat een automaat van niveau 1 beslisbaar is in tijd $O(n^4)$ aangezien het nemen van de doorsnede van twee automaten in tijd $O(n^4)$ kan, met n het aantal toestanden van de grootste automaat.

6.3.4 Level 2

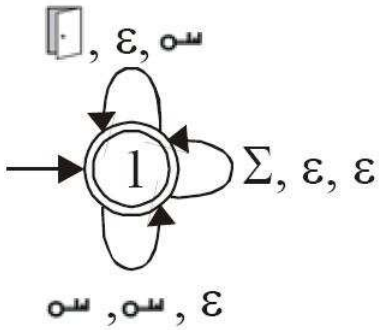
Om het geheel weer een stapje moeilijker te maken, gaan sleutels verdwijnen van het moment we ze gebruikt hebben. We kunnen dus net zo vaak een deur door als dat we sleutels hebben want na ons valt een deur steeds terug in het slot.

Deze voorwaarde kunnen we niet meer voorstellen met een eindige automaat[1]. Wel kunnen we deze voorwaarde voorstellen als een zogenaamde Keller-automaat of pushdown-automaat. Hierbij pushen we alle gevonden sleutels op een stack en voor elke deur die we passeren, poppen we één sleutel van de stack. Deze pushdown-automaat noemen we G' en staat in figuur 6.3. De taal, aanvaard door deze pushdown-automaat, voldoet aan onze nieuwe voorwaarde.

Avonturen van deze moeilijkheidsgraad hebben oplossingen in de doorsnede van de talen van de kaart (M) enerzijds en de voorwaarden G', D en T anderzijds.
$L(A_2) = L(M) \cup L(G') \cup L(D) \cup L(T)$.

De taal, aanvaard door de pushdown-automaat G', is een contextvrije taal. Deze taal is niet regulier aangezien we hem niet kunnen voorstellen met een eindige automaat. De andere automaten (M, D en T) zijn eindige automaten en aanvaarden reguliere talen. Deze zijn ook steeds contextvrij. De doorsnede van contextvrije talen is ook contextvrij. We kunnen dus zeggen dat avonturen van level 2 van Chomsky type 2 zijn. Het is in tijd $O(n^3)$ beslisbaar of avonturen van dit level een oplossing hebben.

[1]Dit wordt in [30] bewezen met behulp van het pumping lemma voor reguliere talen.



Figuur 6.3: De pushdown-automaat voor G'

6.3.5 Level 3

In deze moeilijkheidsklasse gaan we niet alleen de sleutels tellen maar ook de zwaarden. We kunnen een zwaard vanaf nu slechts één keer gebruiken om een draak te verslaan maar als we terug bij deze draak zouden belanden, is hij weer strijdlustig. Indien we geen zwaard hebben om een draak te verslaan, kunnen we nog steeds in een rivier springen.

Als we deze nieuwe voorwaarde willen modeleren, bemerken we dat pushdown-automaten niet meer voldoende zijn[2]. In plaats van automaten kunnen we wel gebruik maken van de theorie van Petri-netten om aan te tonen dat avonturen van level 3 nog steeds beslisbaar zijn. De lezer wordt verwezen naar [30, 31] voor de werkwijze.

In de Chomsky-hiërarchie horen avonturen van level 3 tot de categorie van recursief opsombare talen (Chomsky 0), in dit geval beslisbaar.

6.3.6 Level 4

De hoogste moeilijkheidsgraad is level 4. Hierbij kan de avonturier geen rivier overzwemmen met een zwaard in de hand en kan hij niet onder een boog doorlopen als hij een sleutel in zijn bezit heeft. Om deze avonturen op te lossen kunnen we bijvoorbeeld gebruik maken van two-counter machines waarbij er getest kan worden of een bepaalde teller op nul staat alvorens een transitie te mogen maken.

Deze avonturen zijn recursief opsombare talen (Chomsky 0) waarvan het niet beslisbaar is of ze oplosbaar zijn.

[2]Te bewijzen met het pumping lemma voor contextvrije talen, zie [30]

6.4 Besluit

Finite state adventures zijn interessant als voorbeeld bij het aanleren van automaten en de Chomsky-hiërarchie. Studenten leren spelenderwijs dat het vinden van een oplossing voor een avontuur moeilijker wordt afhankelijk van de eisen die er gesteld worden. Er moet wel over gewaakt worden dat studenten deze avonturen niet rechtstreeks gelijkstellen aan automaten. Een avontuur bestaat namelijk over een eindige automaat enerzijds en de lijst van voorwaarden anderzijds.

Zoals de auteurs van het artikel [30] reeds hebben aangehaald, reageren studenten zeer positief over FSA's. Het is een leuk alternatief voor zowel de studenten als de docenten.

Hoofdstuk 7

XViz, een visualisatietool voor XPath

7.1 Didactische wenken

XViz is een mogelijkheid die de docenten kunnen aangrijpen om het aanleren van XPath expressies te vergemakkelijken. Deze tool kan helpen bij het grafische voorstellen van XPath expressies en de relaties ancestor/descendant en containment. De studenten kunnen een reeks expressies ingeven en in de resulterende graaf de relaties tussen de expressies aflezen. De studenten zullen hierdoor een beter begrip hebben van deze relaties en XPath expressies.

7.2 XViz

Ben Handy en Dan Suciu ontwikkelden XViz als tool voor database administrators. Het moet hen helpen routinetaken uit te voeren zoals database tuning, performance debugging en vergelijking tussen versies. Door de visuele voorstelling in een graaf en de duidelijke zichtbare relaties voorouder/afstammeling (ancestor/descendant) en containment kunnen in een oogopslag kenmerken van XPath expressies afgelezen worden. De betekenis van deze relaties voor XPath worden verklaard in 7.2.2.

7.2.1 Werkwijze

XViz vertrekt van een XQuery workload, een eindige set van XQueries, in een *.xquery*-bestand. Hieruit worden alle XPath expressies gehaald die vervolgens in een nieuw *.eps*-bestand als graaf uitgetekend worden. Elke knoop komt overeen met een XPath expressie in de workload (of een prefix ervan) en alle pijlen staan voor relaties tussen de expressies. Volle lijnen worden getekend tussen ancestor en descendant en stippellijnen tonen containment aan.

Voorbeeld

Beschouwen we de workload zoals getoond in figuur 7.1 dan krijgen we als uitvoer de graaf van figuur 7.2.

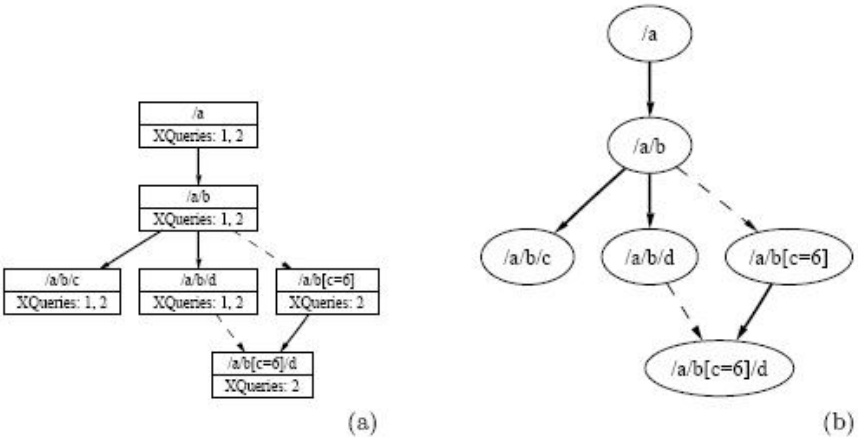
```
Q1:

FOR $x in /a/b
WHERE sum($x/c) > 5
RETURN <result> $x/d </result>

Q2:

FOR $u in /a/b[c=6],
    $v in /a/b
WHERE $u/d > $v/c
RETURN $v/d
```

Figuur 7.1: Een voorbeeld workload



Figuur 7.2: Resultaat van XViz op de workload in figuur 7.1

Hierbij zien we dat voor de expressie $/a/b$ eerst de knoop voor de deelexpressie $/a$ getekend wordt met een volle lijn (van voorouder naar afstammeling) naar de knoop die $/a/b$ voorstelt ($/a \ll /a/b$). $/a/b/c$ En $/a/b/d$ zijn hier dan weer afstammelingen van terwijl alle resultaten van de expressie $/a/b[c = 6]$ ook in de resultaten van $/a/b$ zitten ($/a/b \supseteq /a/b/[c = 6]$). De gebruiker krijgt nu ook zicht op de eerst onduidelijke relaties tussen $/a/b/d$, $/a/b[c = 6]$ en $/a/b[c = 6]/d$.

7.2.2 Relaties

Containment en ancestor/descendant relaties worden semantisch bekeken, de gebruikte algoritmen voor het bepalen van deze relaties zijn dus onafhankelijk van de gebruikte syntactische notatie. Duiden we met p een XPath expressie aan en met t een XML boom, dan staat $p(t)$ voor de set van knopen die het resultaat zijn van de evaluatie van p op t. Hierbij nemen we aan dat elke evaluatie in de wortel van de boom start. Duiden we knopen in t aan met $x, y, ...$ dan zeggen we dat $x \ll y$ als x een voorvader (ancestor) is van y.

Ancestor/descendant ($\ll$)

Voor XPath expressies kunnen we de ancestor/descendant relatie definiëren als volgt: de expressie p' is een ancestor van p als voor alle bomen t en voor alle knopen y van $p(t)$ geldt dat er een knoop x uit $p'(t)$ bestaat die een voorvader is van y ofwel:

$$p' \ll p \Leftrightarrow \forall \text{ XML boom } t \text{ en } \forall y \in p(t) : \exists x \in p'(t) : x \ll y$$

Containment ($\supseteq$)

We zeggen dat een XPath expressie p' de expressie p bevat als voor alle bomen t geldt dat de set knopen die $p'(t)$ teruggeeft alle knopen van $p(t)$ bevat ofwel:

$$p' \supseteq p \Leftrightarrow \forall \text{ XML boom } t : p'(t) \supseteq p(t)$$

We kunnen de containment relatie en de ancestor/descendant relatie reduceren naar elkaar als volgt:

$$p' \ll p \Leftrightarrow p'//* \supseteq p$$

$$p' \supseteq p \Leftrightarrow p'/a \ll p/a/*$$

Equivalentie ($\equiv$)

We kunnen de definitie van equivalentie van XPath expressies afleiden van de definitie van containment:

$$p \equiv p' \Leftrightarrow p \supseteq p' \text{ en } p' \supseteq p$$

Waarbij a eender welke name-tag kan zijn die niet voorkomt in p'.

7.2.3 Het programma

Na het lezen van deze paper [32] was het een teleurstelling te ontdekken dat er twee jaar later geen werkende versie te vinden is. Op de website van Dan Suciu aan de universiteit van Washington wordt de programmacode van het XViz project ter beschikking gesteld. Deze gaf na compilatie echter geen resultaten bij het uitvoeren van het voorbeeld zoals het in de tekst uitgelegd staat. Ook na het bestuderen van de code is het niet gelukt om uitvoer te verkrijgen.

7.3 Besluit

XViz is een zeer interessante tool voor zowel database administrators als studenten bij het werken met XPath expressies. In een oogopslag worden anders moeilijk zichtbare relaties tussen XPath expressies getoond waardoor de database beheerder zijn opdracht kan vereenvoudigen en de student een beter inzicht verwerft in XPath.

Hoofdstuk 8

Besluit van deze thesis

We hebben aangetoond dat het belangrijk is voor docenten om steeds de didactische basisprincipes in het oog te houden. We hebben verschillende onderwerpen uit de cursus theoretische informatica besproken alsook enkele mogelijke uitbreidingen. Verschillende van deze uitbreidingen kunnen enkel gerealiseerd worden door integratie van de theoretische informatica met een ander vak uit het curriculum informatica.

De uitbreiding van de tool JFLAP biedt studenten een *aanschouwelijk* beeld van het CYK parsing algoritme en de aangeboden oefeningen zorgen ervoor dat de studenten *actief* bezig blijven. Ook XViz is een tool die de docenten kan helpen bij het duidelijker voorstellen van de theorie, in dit geval XPath expressies.

De vergelijking tussen de Thomspon- en Glushkov-automaten zorgt voor een betere *integratie* van de aanwezige kennis (Thompson) en te verwerven kennis (Glushkov).

De toepassingen op boomautomaten én de pattern matching algoritmes, die gebruik maken van stringautomaten, zijn goede voorbeelden van de *integratie* van onderwerpen van verschillende vakken. Deze topics zijn ideaal om in een keuzevak aan te bieden. Zo kan er *gedifferentieerd* worden tussen studenten zich willen verdiepen in de theoretische informatica en de studenten die enkel een minimale basis moeten meekrijgen.

Een zeer interessante manier om *belangstelling* te wekken voor de onderwerpen die aangeboden worden in de cursus is het voorbeeld van finite state adventures. Studenten leren spelenderwijs dat het vinden van een oplossing voor een avontuur moeilijker wordt afhankelijk van de eisen die er gesteld worden. Deze aanpak steunt ook op het principe van de *geleidelijkheid*.

Bibliografie

[1] F. Neven: *Trajectboek Theoretische Informatica*, Limburgs Universitair Centrum, 2004.

[2] M. Gyssens: *Cursus Theoretische Informatica*, Limburgs Universitair Centrum, 2000.

[3] M. Sipser: *Introduction to the theory of computation*, PWS Publishing Company, 1997.

[4] J. Tielemans: *Psyhodidactiek en het onderwijs*, Garant Leuven, 1999.

[5] *JFLAP*. Beschikbaar op: http://www.cs.duke.edu/∼rodger/tools/JFLAP

[6] J. Van Den Bussche: *Cursus Compilers en programmeeromgevingen*, Limburgs Universitair Centrum, 2002.

[7] D. Giammaresi, J.-L. Ponty, D. Wood: *A reexamination of the Glushkov and Thompson constructions*, unpublished manuscript, 1998

[8] A. Brüggemann-Klein , D. Wood: *One-unambiguous regular languages*, Information and Computation, v.140 n.2, p.229-253, Feb. 1, 1998.

[9] W. Jans, W. Janssen: *Seminarie databases: One-unambiguous reguliere talen*, transnationale Universiteit Limburg, november 2004.

[10] V.M. Glushkov: *The abstract theory of automata*, Russian Mathematical Surveys, 16:1-53, 1961.

[11] W3C: *XML specificatie*. Beschikbaar op: http://www.w3.org/XML

[12] K. Luyten: *Inleiding tot XML en aanverwante specificaties, Technologie van Multimediasystemen en -software*, Limburgs Universitair Centrum, 2001.

[13] M. Gyssens en S. Vansummeren: *Sylabys logica en databases*, Hoofdstuk 3 XPath, Limburgs Universitair Centrum, 2005.

[14] G. Gottlob, C. Koch, R. Pichler: *Efficient Algorithms for Processing XPath Queries*, VLDB 2002: 95-106.

[15] S. Vansummeren: *Type inferentie in XML, de relationele algebra en XDuce*, Thesis, Limburgs Universitair Centrum, 2001.

[16] F. Neven: *Automata Theory for XML Researchers*, SIGMOD Record 31(3): 39-46 (2002).

[17] F. Neven: *Design and Analysis of Query Languages for Structured Documents (A Formal and Logical Approach)*, PhD thesis, Limburgs Universitair Centrum, 1999.

[18] A. Brüggemann-Klein, M. Murata, and D. Wood: *Regular tree languages over non-ranked alphabets*, Unpublished manuscript, 1998.

[19] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison and M. Tommasi: *Tree automata techniques and applications*, chapter 1, 2002. Handboek beschikbaar op: http://www.grappa.univ-lille3.fr/tata/

[20] T. Schwentick: *XPath query containment*, SIGMOD Record 33(1): 101-109, 2004.

[21] P. Linz: *An Introduction To Formal Languages And Automata*, Handbook, third edition, 2000.

[22] J. Boyer: *Canonical XML*, Version 1.0, March 2001. Beschikbaar op: http://www.w3.org/TR/2001/REC-xml-c14n-20010315

[23] D.R.Raymond, F.W.Tompa, and D.Wood: *From Data Representation to Data Model: Meta-Semantic Issues in the Evolution of SGML*, Computer Standards & Interfaces, 18 (1996), 25-36.

[24] G. Miklau , D. Suciu: *Containment and equivalence for an XPath fragment*, Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, June 03-05, 2002, Madison, Wisconsin.

[25] F. Neven: *Automata, Logic, and XML*, J.C.Bradfield(Ed.): Computer Science Logic, Lecture notes in computer science, 2-26, Springer-Verlag, 2002.

[26] National institute of standards and technology: *Dictionary of Algorithms and Data Structures*. Beschikbaar op: http://www.nist.gov/dads/

[27] B.W. Watson: *Cursus object georiënteerd programmeren, Multiple Keyword Exact Pattern Matching*. Beschikbaar op: http://www.win.tue.nl/∼watson/2R080/opdracht/2r080_Opdracht.htm

[28] C. Charras and T. Lecroq: *Hanbook of exact string matching algorithms*, King's College London Publications, 2004.

[29] D. Eppstein: *Design and analysis of algorithms*, lecture notes, 1996. Beschikbaar op: http://www.ics.uci.edu/∼eppstein/161/960227.html

[30] W. Brauer, M. Holzer, B. König and S. Schwoon: *The theory of finite state adventures*, EATCS Bulletin, 79:230-237, February 2003.

[31] M. Meeser: *Eine Komplexitätstheorie von Adventure-Spielen*, Seminar "Perlen der Theoretischen Informatik" AG Meyer auf der Heide, Wintersemester 2004/2005.

[32] B. Handy and D. Suciu: *XViz: A tool for visualising XPath expressions*, Z. Bellahsne, A.B. Chaudhri, E. Rahm, M. Rys and R. Unland (Eds.), Database and XML technologies, Lecture notes in computer science, 134-148, Springer, 2003.