

# Studentenhandleiding

## Voortgezette informatica (TVIT)

Docenten: Jansen (Jan Martin)

Versie: 2024.0

# 1. Inleiding

In het vak Voortgezette Informatica worden een aantal theoretische grondslagen behandeld die een belangrijke rol spelen binnen de informatica.

De begrippen algoritme, computer en programma en het verband ertussen worden behandeld. Wat moet een computer minimaal kunnen en welke constructies moeten er minimaal in een programmeertaal zitten om praktisch bruikbaar te zijn?

Er worden een aantal fundamentele computer modellen zoals eindige automaten, stapelautomaten en Turing machines behandeld en er wordt beschreven welke mogelijkheden en beperkingen deze modellen hebben.

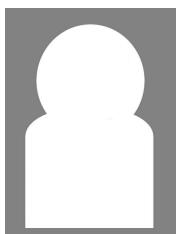
Deze handleiding is een werk in uitvoering. Een groot deel van de lesstof is (ook) te vinden in deze handleiding, maar er wordt soms ook naar de lessen en presentaties verwezen. Verder kan het zijn dat je tijdens de uitvoering van de cursus nog een bijgewerkte versie van de handleiding krijgt.

## 1.1 Toetsing

Schriftelijke toets 3 uur na afloop, inspanningsverplichting voor huiswerkopgaven.

## 1.2 Docenten

Onderstaande docenten zijn betrokken bij deze uitvoering van deze cursus.



**Jansen (Jan Martin) (coordinator)**

Email: jm.jansen.04@mindef.nl

Kamer: E.2047

## 2. Bijeenkomsten

In dit hoofdstuk vind je een overzicht van de geplande bijeenkomsten van de cursus. Mocht je door onvoorzien omstandigheden een keer niet aanwezig kunnen zijn, dan kun je hier terugvinden welke stof er tijdens die bijeenkomst besproken is.

### Overzicht

Nr	Bijeenkomst
1	<b>Eindige automaten en reguliere expressies 1</b> Dinsdag 11 juni 2024
2	<b>Eindige automaten en reguliere expressies 2</b> Woensdag 12 juni 2024
3	<b>Pushdown automaten en grammatica's 1</b> Dinsdag 18 juni 2024
4	<b>Pushdown automaten en grammatica's 2</b> Woensdag 19 juni 2024
5	<b>Turingmachines 1</b> Dinsdag 25 juni 2024
6	<b>Turingmachines 2</b> Woensdag 26 juni 2024
7	<b>Uitloop 1</b> Dinsdag 2 juli 2024
8	<b>Talen afbeelden 1</b> Woensdag 3 juli 2024
9	<b>Talen afbeelden 2</b> Dinsdag 9 juli 2024
10	<b>Uitloop 2</b> Woensdag 10 juli 2024
11	<b>Recursie en Iteratie 1</b> Dinsdag 16 juli 2024
12	<b>Recursie en Iteratie 2</b> Woensdag 17 juli 2024

## 2.1 Les 1: Eindige automaten en reguliere expressies 1

Te behandelen onderwerpen:	Talen, Deterministische eindige automaten, Non-deterministische eindige automaten, Reguliere expressies
Geplande activiteiten:	<ul style="list-style-type: none"> <li>• Cursusoverzicht</li> <li>• Inleiding Informatica</li> <li>• Theorie eindige automaten</li> <li>• Eindige automaat maken</li> <li>• Eindige automaat analyseren</li> </ul>

## 2.2 Les 2: Eindige automaten en reguliere expressies 2

Te behandelen onderwerpen:	Equivalentie non-deterministische en deterministische automaten, Equivalentie reguliere expressies en eindige automaten
Geplande activiteiten:	<ul style="list-style-type: none"> <li>• Bespreken huiswerk</li> <li>• Uitleg deterministisch maken automaten</li> <li>• Oefenen deterministisch maken</li> <li>• Uitleg reguliere expressie omzetten</li> <li>• Oefenen reguliere expressie omzetten</li> </ul>

## 2.3 Les 3: Pushdown automaten en grammatica's 1

Te behandelen onderwerpen:	Grammatica's, Pushdown automaten
Geplande activiteiten:	<ul style="list-style-type: none"> <li>• Bespreken huiswerk</li> <li>• Gelijk delen</li> <li>• Uitleg pushdown automaten</li> <li>• Oefenen pushdown automaten</li> <li>• Intro grammatica's</li> </ul>

## 2.4 Les 4: Pushdown automaten en grammatica's 2

Te behandelen onderwerpen:	Pushdown automaten, Grammatica's, Expressiebomen
Geplande activiteiten:	<ul style="list-style-type: none"> <li>• Bespreken huiswerk</li> <li>• Rekenkundige expressies</li> <li>• Oefenen met grammatica's</li> </ul>

## 2.5 Les 5: Turingmachines 1

Te behandelen onderwerpen:	Turingmachines
Geplande activiteiten:	<ul style="list-style-type: none"> <li>• Herhalen en bespreken huiswerk</li> <li>• Herhaling Ontleden</li> <li>• Uitleg Turingmachines</li> <li>• Oefening Turingmachine Palindroom</li> </ul>

## 2.6 Les 6: Turingmachines 2

Te behandelen onderwerpen:	Turingmachines, Halting probleem
Geplande activiteiten:	<ul style="list-style-type: none"> <li>• Bespreken huiswerk</li> <li>• Het halting probleem</li> <li>• Oefenen</li> </ul>

## 2.7 Les 7: Uitloop 1

Te behandelen onderwerpen:	Hogere orde functies, Getalrepresentaties
Geplande activiteiten:	<ul style="list-style-type: none"> <li>• Herhalen automaten en expressies</li> <li>• Herhalen pushdown automaten en grammatica's</li> <li>• Herhalen Turingmachines</li> </ul>

## 2.8 Les 8: Talen afbeelden 1

Te behandelen onderwerpen:	Processoren, Virtuele machines, IJVM
Geplande activiteiten:	<ul style="list-style-type: none"> <li>• Uitleg Processoren</li> <li>• Uitleg Virtual machines</li> <li>• Introductie IJVM</li> <li>• IJVM Als rekenmachine</li> <li>• Implementatie conditions and loops</li> <li>• Oefenen conditionals</li> <li>• Uitleg functiecalls</li> <li>• Oefenen functiecalls</li> </ul>

## 2.9 Les 9: Talen afbeelden 2

Te behandelen onderwerpen:	JMVM, Callstacks, Heaps
Geplande activiteiten:	<ul style="list-style-type: none"> <li>• Bespreken huiswerk</li> <li>• Herhaling IJVM functies</li> <li>• Oefenen functies vertalen</li> <li>• Uitleg heap-geheugen</li> <li>• Oefenen gebruik heap</li> </ul>

## 2.10 Les 10: Uitloop 2

Te behandelen onderwerpen:	-
Geplande activiteiten:	<ul style="list-style-type: none"> <li>• Geen activiteiten gepland</li> </ul>

## 2.11 Les 11: Recursie en Iteratie 1

Te behandelen onderwerpen:	Recursieve algoritmes
----------------------------	-----------------------

Geplande activiteiten:	<ul style="list-style-type: none"><li>• Huiswerk bespreken</li><li>• Uitleg recursie</li><li>• Iteratieve torens van Hanoi</li><li>• Strategie recursie</li><li>• Oefenen recursie</li></ul>
------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

## 2.12 Les 12: Recursie en Iteratie 2

Te behandelen onderwerpen:	Recursieve datastructuren
Geplande activiteiten:	<ul style="list-style-type: none"><li>• Herhaling en huiswerk</li><li>• Uitleg recursieve structuren</li><li>• Bestanden zoeken</li><li>• Tentamenvoorbereiding</li></ul>

# 3. Theorie

In dit hoofdstuk vind je de in de bijeenkomsten behandelde theorie. Deze kun je gebruiken voor zelfstudie en als naslagwerk.

## 3.1 Reguliere talen en eindige automaten

Talen en zeker programmeertalen hebben een duidelijke structuur. Eindige automaten en grammatica's zijn manieren om die structuur te beschrijven.

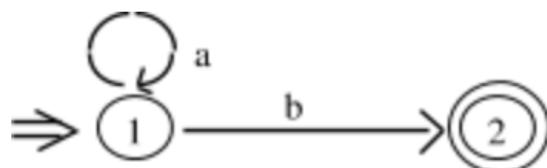
Definities:

- een taal is een verzameling strings
- een string is een rij letters
- een alfabet is een verzameling letters

Voorbeelden:

- $\{ a, b \}$  is een alfabet
- aa is een string
- aaaaab is een string
- $\epsilon$  is een string (een speciaal geval: de string van nul letters)
- $\{ b, ab, aab, aaab, \dots \}$  is een taal

De bovenstaande taal heeft een nogal simpele structuur: enkele a's gevolgd door een b. Zulke vrij eenvoudige talen kunnen met een eindige automaat gegenereerd worden. Voor de voorbeeldtaal voldoet de volgende eindige automaat:



Deze automaat bestaat uit toestanden (aangegeven met een cirkel) en overgangen (aangegeven met een pijl). De toestanden hebben als labels 1 en 2. Er zijn overgangen van 1 naar 1 met als label a en van 1 naar 2 met als label b. Het symbool  $=>$  geeft aan dat toestand 1 de begintoestand is. De dubbele rand om toestand 2 geeft aan dat dit een eindtoestand is. Elke automaat heeft altijd 1 begintoestand, maar kan meerdere eindtoestanden hebben.

Een automaat werkt als volgt:

- hij begint in de begintoestand

- hij maakt telkens een overgang
- hij stopt in een eindtoestand

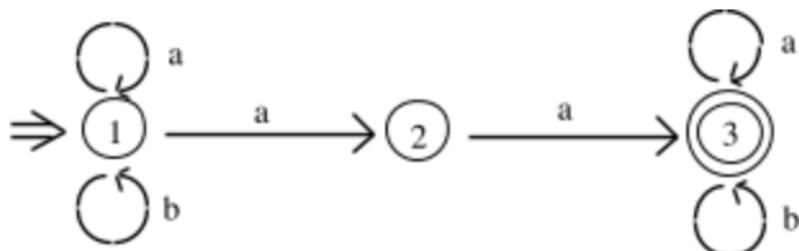
Belangrijk hierbij is dat de automaat mag stoppen in een eindtoestand, maar ook door kan gaan. De labels van de gemaakte overgangen vormen achter elkaar een string. Alle strings die de automaat zo kan maken vormen tezamen de taal van de automaat.

Voorbeeld:



De taal van deze automaat is de verzameling van alle strings die bestaan uit een oneven aantal a's.

Voorbeeld:



De taal van deze automaat is de verzameling van alle strings bestaande uit a's en b's, waarin aa voorkomt als deelstring.

Met deze automaat is iets bijzonders aan de hand, want hij kan de string aaa op 2 manieren maken:

$1 \rightarrow 1 \rightarrow 2 \rightarrow 3$   
 $1 \rightarrow 2 \rightarrow 3 \rightarrow 3$

Dit is op zich een ongewenst gedrag, want de automaat moet als het ware kiezen en als hij aaa zou proberen te maken door te beginnen met  $1 \rightarrow 1 \rightarrow 1$  dan lukt het niet meer om aaa te maken.

Definitie: een eindige automaat is **deterministisch** als hij elke string op hoogstens 1 manier kan maken.

Een eindige automaat die een bepaalde string op meer dan 1 manier kan maken, noemen we non-deterministisch. Zo'n automaat is te herkennen aan het feit dat hij een toestand bevat van waaruit je met een bepaalde letter naar meer dan 1 toestand kan gaan. In het voorbeeld gebeurt dit bij toestand 1 want met de letter a kun je zowel naar toestand 1 als naar toestand 2 gaan.

## 3.2 Een eindige automaat bouwen

Een aardige manier om een deterministische eindige automaat te construeren voor een bepaalde taal is om elke toestand iets te laten onthouden van de tot nog toe gegenereerde letters.

Voorbeeld: maak een automaat voor de taal van alle strings bestaande uit a's en b's waarin aa voorkomt. Van belang om te weten is nu

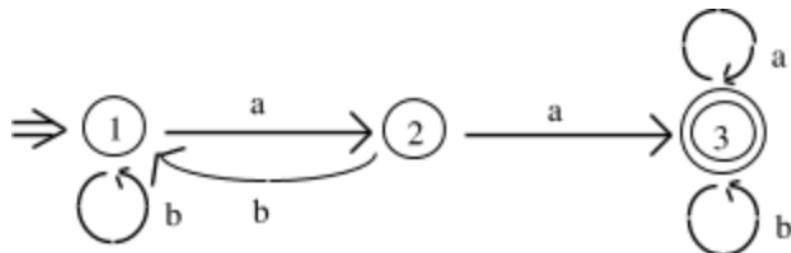
P: we hebben aa al gehad Q: de laatste letter was een a

Voor de automaat zijn de volgende toestanden van belang:

- 1)  $\neg P \quad \neg Q$
  - 2)  $\neg P \quad Q$
  - 3)  $P$

We zouden toestand 3 ook weer op kunnen splitsen in  $P \rightarrow Q$  en  $P \wedge Q$  maar dit hoeft niet omdat we als we al gehad hebben er willekeurig wat achter kunnen zetten. Het verschil tussen wel of niet als laatste een speelt geen enkele rol meer.

## De automaat wordt:

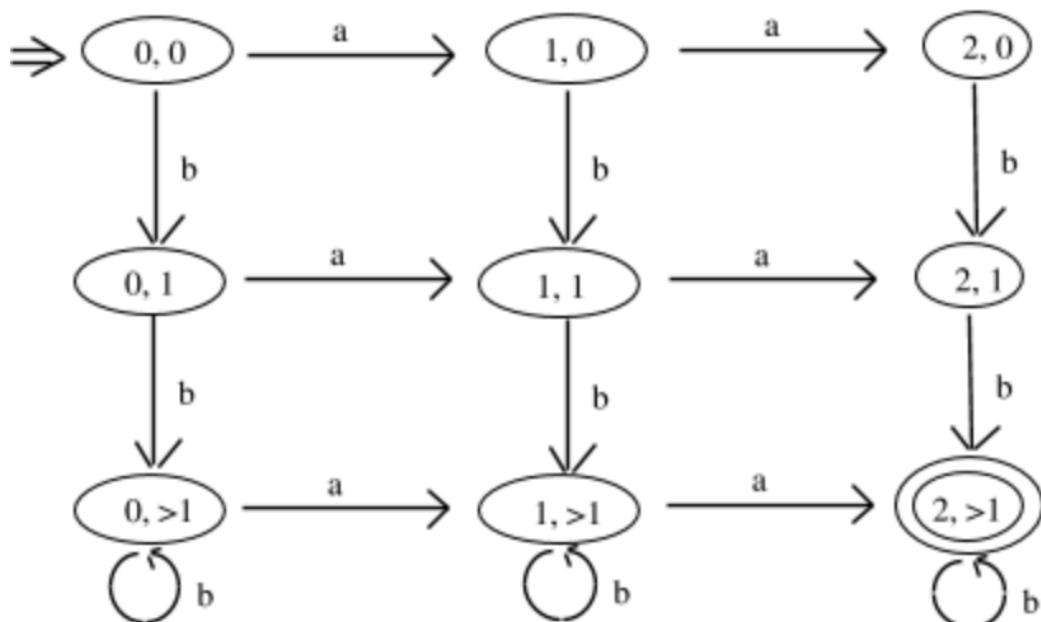


In het begin hebben we het lege woord. Dus we hebben aa nog niet gehad en de laatste letter was geen a. De begintoestand is dus 1. Als er in toestand 1 een a komt, dan hebben we nog steeds aa niet gehad maar de laatste letter is wel een a, daarom wordt de toestand 2. Als er in toestand 1 een b komt, dan is de laatste letter geen a en blijven we dus in toestand 1. Zo kun je ook de andere overgangen berecdeneren. Dat toestand 3 eindtoestand is, is omdat we dan aa gehad hebben.

Voorbeeld: maak een automaat voor de taal van alle strings bestaande uit a's en b's die precies 2 a's en minstens 2 b's bevatten.

We moeten nu weten of we 0, 1 of 2 a's gehad hebben en of we 0, 1 of meer dan 1 b gehad hebben.

De automaat wordt:



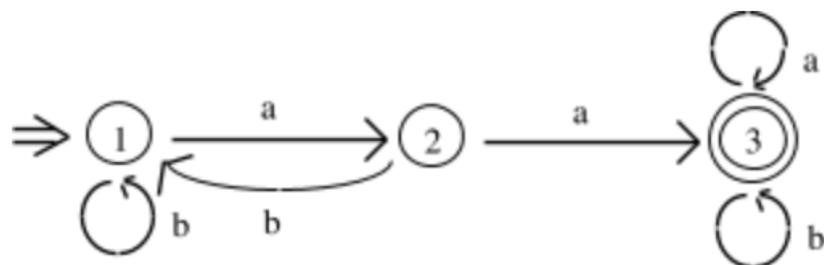
De labels van de toestanden hebben de volgende betekenis: het eerste cijfer geeft het aantal a's aan en het tweede cijfer geeft het aantal b's dat we al gehad hebben. Vanuit de toestand  $z_0$  is er geen overgang

met de letter a, omdat we geen strings mogen genereren met meer dan 2 a's.

### 3.3 Determinisme

Een deterministische eindige automaat kunnen we gebruiken om te testen of een string tot zijn taal behoort, want bij elke string hoort er dan hoogstens 1 pad en dus hoogstens 1 toestand waar we met deze string uitkomen. Als die resulterende toestand een eindtoestand is dan hoort de string tot de taal en anders niet.

Voorbeeld:

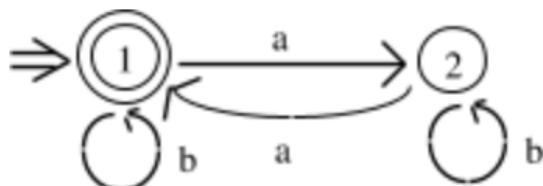


De string aaab heeft als pad: 1  $\rightarrow$  2  $\rightarrow$  3  $\rightarrow$  3  $\rightarrow$  3. Omdat we in 3 uitkomen en 3 eindtoestand is kan de automaat de string maken. De string bba heeft als pad: 1  $\rightarrow$  1  $\rightarrow$  1  $\rightarrow$  2. Nu komen we in 2 uit en 2 is geen eindtoestand; daarom kan de automaat deze string niet maken.

Voorbeeld: maak een eindige automaat om te testen of een string bestaat uit een even aantal a's en een willekeurig aantal b's.

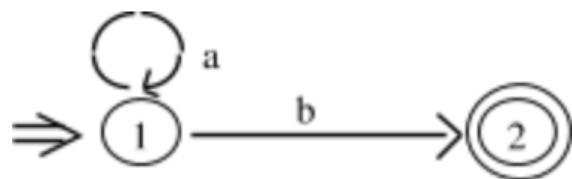
De bijbehorende automaat heeft 2 toestanden:

- 1) we hebben een even aantal a's
- 2) we hebben een oneven aantal a's

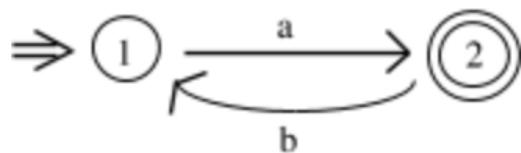


### 3.4 Reguliere expressies

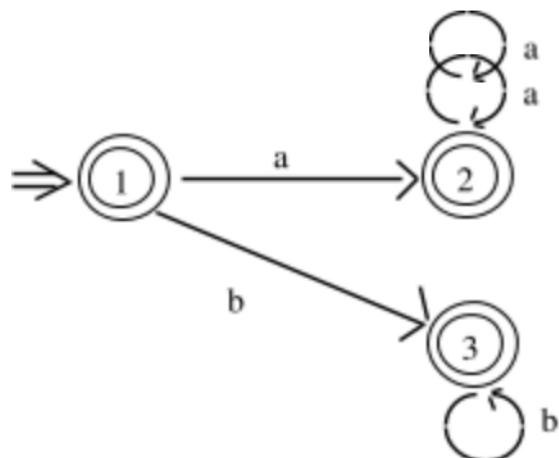
De taal van een eindige automaat heeft doorgaans een eenvoudige herhalende structuur zoals de volgende voorbeelden laten zien:



Taal: b, ab, aab, aaab ... Patroon:  $a^*b$



Taal: a, aba, ababa, abababa ... Patroon:  $(ab)^*$

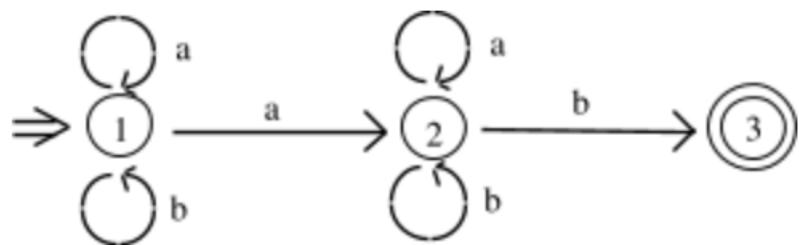


Taal: , a, aa, aaa ..., b, bb, bbb ... Patroon:  $a^* \mid b^*$

Bij de automaten hebben we het patroon waaraan hun strings voldoen vermeld. Zo'n patroon heet ook wel een **reguliere expressie**. Hij kan bestaan uit letters, keuze en herhaling. De keuze tussen x en y geven we aan als  $x \mid y$ . Een herhaling van o of meer keer x geven we aan als  $x^*$ . De conventie is dat herhaling (\*) sterker bindt dan keuze (|). Dus  $a^* \mid b^*$  betekent  $(a^*) \mid (b^*)$  en is iets heel anders dan  $(a \mid b)^*$ .

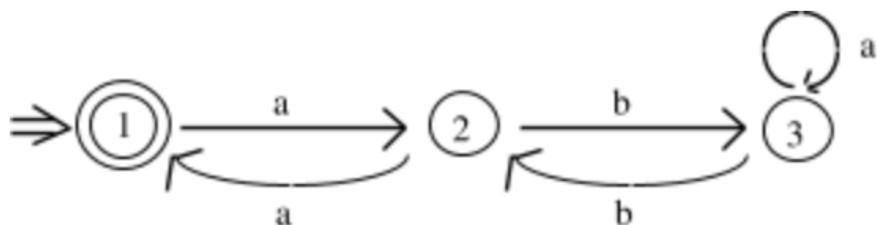
Het is meestal niet zo moeilijk om bij een reguliere expressie een equivalente automaat te construeren.

Voorbeeld: maak een automaat met als taal  $(a \mid b)^*a(a \mid b)^*$



Andersom dus van een automaat een reguliere expressie geven is doorgaans ook niet al te ingewikkeld. Waar het op neer komt is dat je toestanden als het ware wegwerkt.

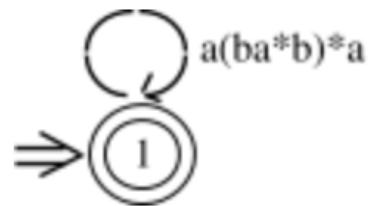
Voorbeeld: bepaal een reguliere expressie voor de taal van de automaat



Dit lijkt een onhandelbaar ding met al die pijlen heen en weer terug. Toch is de situatie in de buurt van toestand 3 vrij eenvoudig. We kunnen toestand 3 wegwerken als we een reguliere expressie langs een pijl zetten:

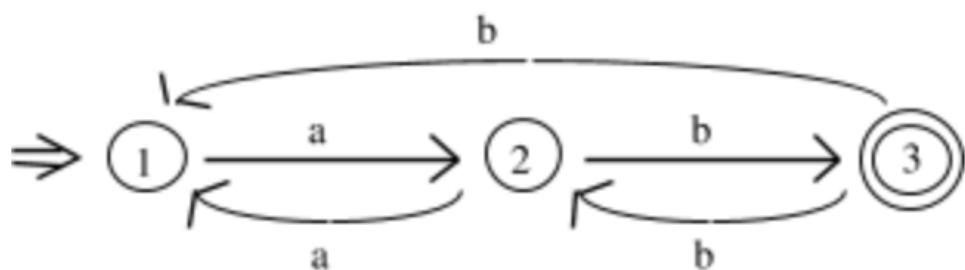


Na deze manipulatie is ook de situatie rond toestand 2 eenvoudig geworden. We werken dus nu toestand 2 weg.

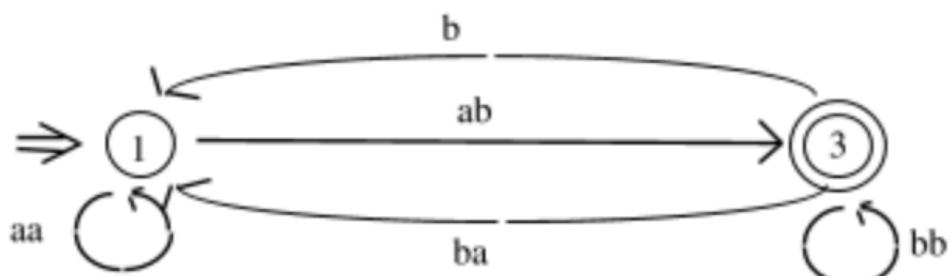


Dus de reguliere expressie is:  $(a(ba^*b)^*)^*$

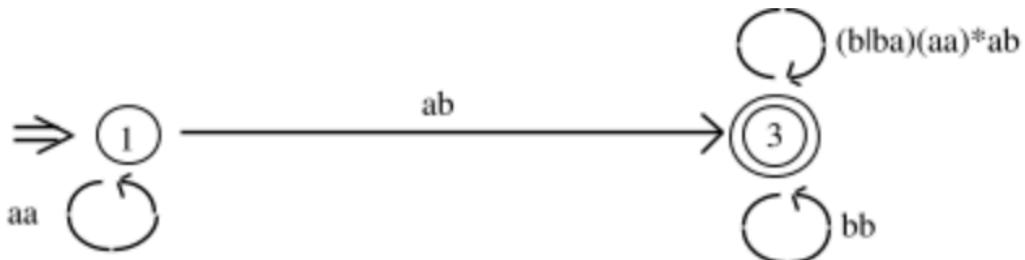
Nog een voorbeeld:



Nu kunnen we toestand 2 wegwerken door elke overgang naar 2 met 1 stap te verlengen:



Alle overgangen naar toestand 1 kunnen we op dezelfde manier behandelen, waardoor we toestand 1 gedeeltelijk wegwerken.



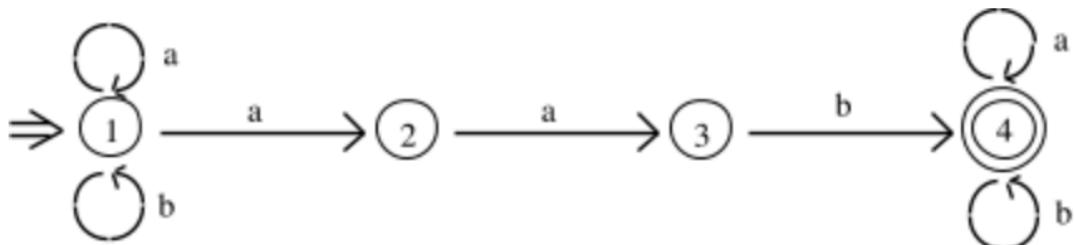
De af te lezen reguliere expressie is dan:  $(aa)^* ab ( bb \mid (blba)(aa)^*ab )^*$ .

### 3.5 Deterministisch maken van een automaat

Om strings te herkennen is een deterministische eindige automaat handig, maar vaak is het eenvoudiger om een non-deterministische automaat ervoor te construeren.

Stel we willen een eindige automaat maken voor alle strings bestaande uit a's en b's waarin de string aab voorkomt.

Een non-deterministische automaat hiervoor is:



Een deterministische automaat hiervoor maken is lastiger. Maar het blijkt dat er een algoritme is om van een non-deterministische automaat een deterministische automaat te maken, die precies dezelfde taal genereert. De deterministische automaat bootst de non-deterministische automaat na. De string aaa heeft de volgende paden in de non-deterministische automaat:

```

1 --> 1 --> 1 --> 1
1 --> 1 --> 1 --> 2
1 --> 1 --> 2 --> 3
  
```

Dus vanuit 1 kom je met aaa in 1, 2 of 3. De deterministische automaat heeft hoogstens 1 pad voor aaa. De toestand die uiteindelijk bereikt wordt zullen we 1, 2, 3 noemen, omdat hij overeenkomt met paden in de oorspronkelijke automaat die in 1, 2 of 3 uitkomen. Kijken we nu naar de string aaab, dan kan die in de oorspronkelijke automaat uitkomen in 1 of 4. Hiervoor hoeven we alleen te kijken hoe je de paden van aaa kunt verlengen met een overgang met een b. We hoeven dus alleen te kijken naar de eindtoestanden van aaa en hoe je van daaruit verder kan met een b.

In een schema:

```

1 - b -> 1
2 - b ->
3 - b -> 4
  
```

-----  
 $1,2,3 - b \rightarrow 1,4$

In de nieuw te maken deterministische automaat komt dus de overgang van 1,2,3 naar 1,4 voor met de letter b.

Omdat er maar 1 begintoestand is, begint de nieuwe automaat op dezelfde manier als de oorspronkelijke:



Beginnend vanuit de unieke begintoestand krijgen we:

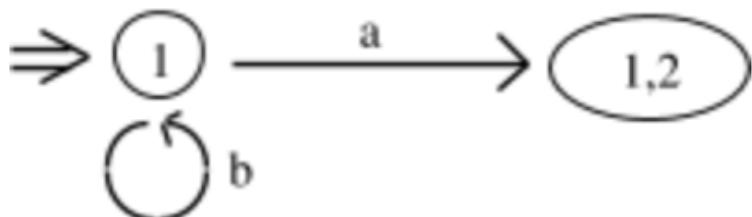
$$1 - a \rightarrow 1$$

$$1 - a \rightarrow 2$$

-----  
 $1 - a \rightarrow 1,2$

$$1 - b \rightarrow 1$$

Dit levert de eerste stap:



Vanuit 1,2 heb je overgangen:

$$1 - a \rightarrow 1$$

$$1 - a \rightarrow 2$$

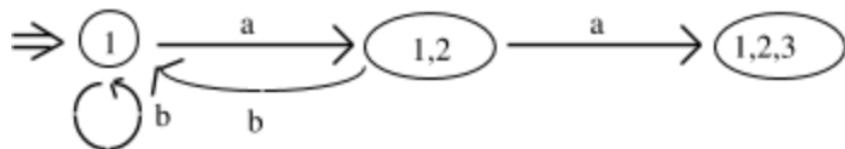
$$2 - a \rightarrow 3$$

-----  
 $1,2 - a \rightarrow 1,2,3$

$$1 - b \rightarrow 1$$

$$2 - b \rightarrow$$

-----  
 $1,2 - b \rightarrow 1$



Nu gaan we kijken wat we vanuit 1,2,3 kunnen doen:

$$1 - a \rightarrow 1$$

$$1 - a \rightarrow 2$$

$$2 - a \rightarrow 3$$

$$3 - a \rightarrow$$

-----

$$1,2,3 - a \rightarrow 1,2,3$$

$$1 - b \rightarrow 1$$

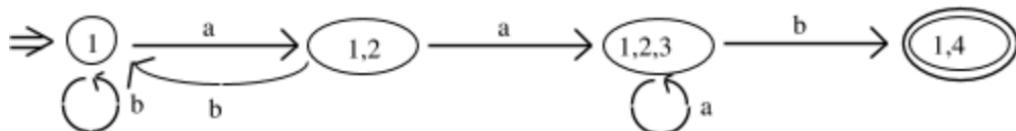
$$2 - b \rightarrow$$

$$3 - b \rightarrow 4$$

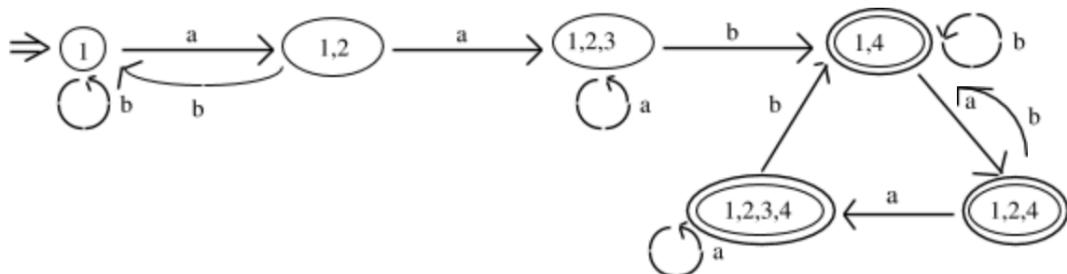
-----

$$1,2,3 - b \rightarrow 1,4$$

Nu is 4 een eindtoestand in de oorspronkelijke automaat, dus 1,4 is ook een eindtoestand in de nieuwe automaat. Het resulteert in:



We kunnen dit procede stug doorzetten en de totale deterministische automaat wordt dan:



### 3.6 Van expressie naar automaat

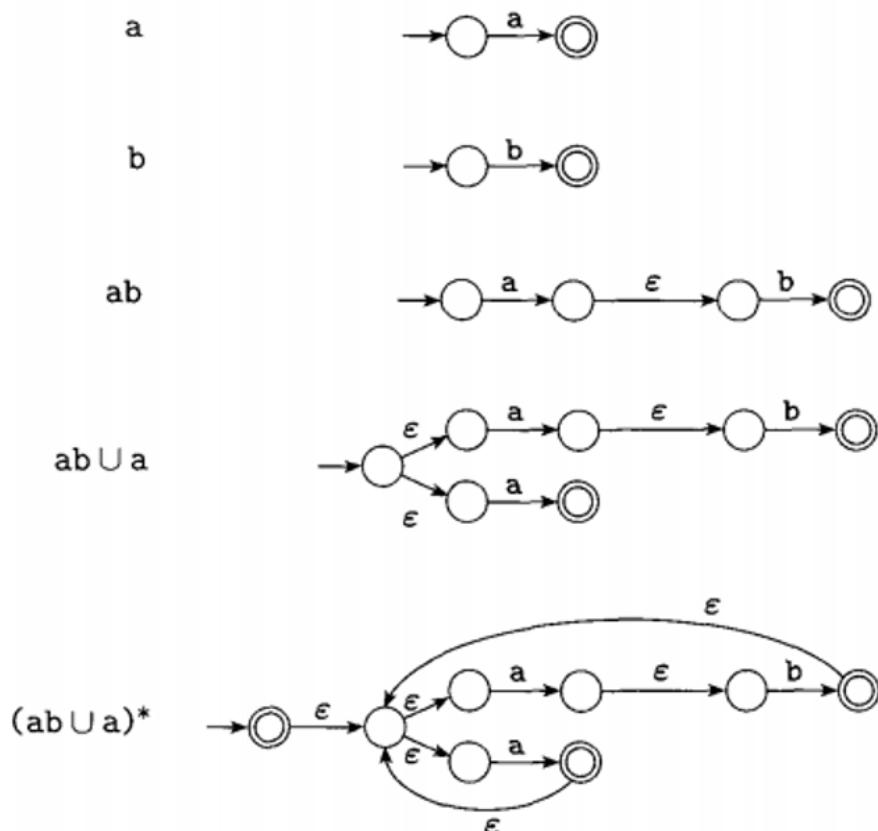
Reguliere expressie zijn handig om compact een taal mee te beschrijven. Ze zijn minder geschikt om een invoer mee te controleren. Eindige automaten werken daar beter voor. Gelukkig kunnen we een reguliere expressie systematisch omzetten in een eindige automaat. Dit doen we door van iedere stukje uit de expressie een deelautomaat te maken en die samen te stellen tot een grotere automaat.

Het samenstellen van complexe automaten uit eenvoudigere doen we door de eindtoestanden van de ene automaat te verbinden met de begintoestand van een andere via een zogenoemde epsilon-overgang.

Dit is een toestandsovergang die gemaakt mag worden door de lege string te “lezen”. Hij kan dus altijd gekozen worden (een automaat met epsilon-overgangen is daardoor altijd niet-deterministisch).

De aanpak is globaal als volgt:

- a wordt pijl met a
  - ab wordt a eps b
  - A | B extra begin en eind (indien nodig) met eps overgangen
  - A\* eps overgang terug naar begin (zorg voor unieke begin en eindtoestand)
- 



### 3.7 Pushdown automaten

Eindige automaten en reguliere expressies kunnen heel veel verschillende talen beschrijven, maar er is een grens aan wat je er mee kunt uitdrukken. Probeer maar eens een eindige automaat te construeren voor de volgende taal voor een alfabet met de symbolen a en b:

Een willekeurig aantal keer het symbool a gevolgd door precies evenveel keer het symbool b. Bijvoorbeeld: ab, aabb of aaaabbbb.

Je zult merken dat het niet lukt om zo'n automaat te maken. Het aantal toestanden dat zo'n automaat nodig heeft is oneindig groot.

Om deze taal te kunnen beschrijven hebben we een krachtiger soort automaat nodig. Een soort automaat waar dit wel mee kan is de zogenaamde pushdown automaat (of stapel-automaat). Dit is een automaat die naast de invoerstring ook gebruikt van een stack (stapel) om te bepalen welke toestandsovergang er gemaakt kan worden.

Een stack is een soort geheugen waarin je symbolen kunt opslaan, maar waarbij je enkel het laatst opgeslagen element kunt zien.

Een pushdown automaat heeft de volgende soorten overgangen:

- s -/- lees s op de invoer (er gebeurt niets met de stack)
- s a/- lees s op de invoer, herken a op stack en verwijder
- s -/a lees s op de invoer, plaats a op stack
- s a/b lees s op de invoer, herken a op stack en vervang door b

Alle stack varianten zijn ook mogelijk met epsilon overgang, bijvoorbeeld:

- - -/a plaats a op stack (lees geen invoer)

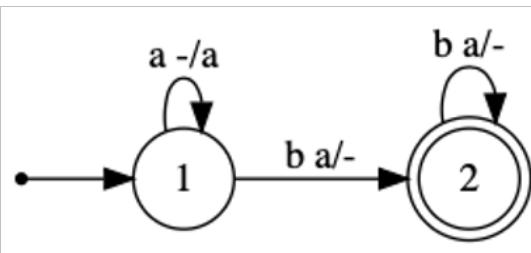
De automaat accepteert als: invoer leeg, stack leeg en in eindtoestand!

Met deze uitgebreide automaat kunnen we taal nu wel herkennen. Voor iedere a in de invoer plaatsen we een symbool op de stack. Bij iedere b die we lezen verwijderen we dat symbool weer. Als we eindigen met een netjes was de input deel van de taal.

```

1 a -/a 1
1 b a/- 2
2 b a/- 2
start 1
end 2

```



## 3.8 Grammatica's

Net als eindige automaten zijn ook reguliere beperkt in de talen die ze kunnen beschrijven. Als we er meer mee uit willen drukken moeten we ze uitbreiden.

Een mogelijke eerste stap is het geven van namen aan deelexpressies. Zo kunnen we bijvoorbeeld een ingewikkelde expressie om strings die getallen voorstellen mee te beschrijven opschrijven als een combinatie van deelexpressies:

```

digit = 0|1|2|3|4|5|6|7|8|9
nzdig = 1|2|3|4|5|6|7|8|9
number = 0 | nzdig digit* [. (0|digit* nzdigit)]

```

Hiermee kunnen we complexe expressie duidelijker opschrijven, maar de verzameling talen die er mee uit kunnen drukken is nog steeds gelijk. We kunnen namelijk door simpelweg de bijbehorende expressies op de plaats van de namen in te vullen de expressie uitschrijven tot de versie zonder namen:

```
0|(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*[.(0| (0|1|2|3|4|5|6|7|8|9)* (1|2|3|4|5|6|7|8|9))]
```

Wanneer we het toestaan dat we in een reguliere expressie met een naam ook toestaan dat de expressie naar zijn eigen naam verwijst (recursie!) kunnen wel meer talen aan.

Kijk bijvoorbeeld weer naar de taal met een willekeurig aantal keer het symbool a gevolgd door precies evenveel keer het symbool b (ab, aabb of aaaabbbb etc.)

Als we die proberen te vatten in een normale reguliere expressie lukt dat niet. De herhalingsoperator \* kan namelijk niet uitdrukken hoe vaak iets herhaald is, en als we het met de operator voor alternatieven | proberen is het aantal mogelijkheden oneindig:

ab | aabb | aaabbb | aaaabbbb | ...

Met een enkele recursieve expressie kunnen dit wel uitdrukken:

$S = ab \mid aSb$

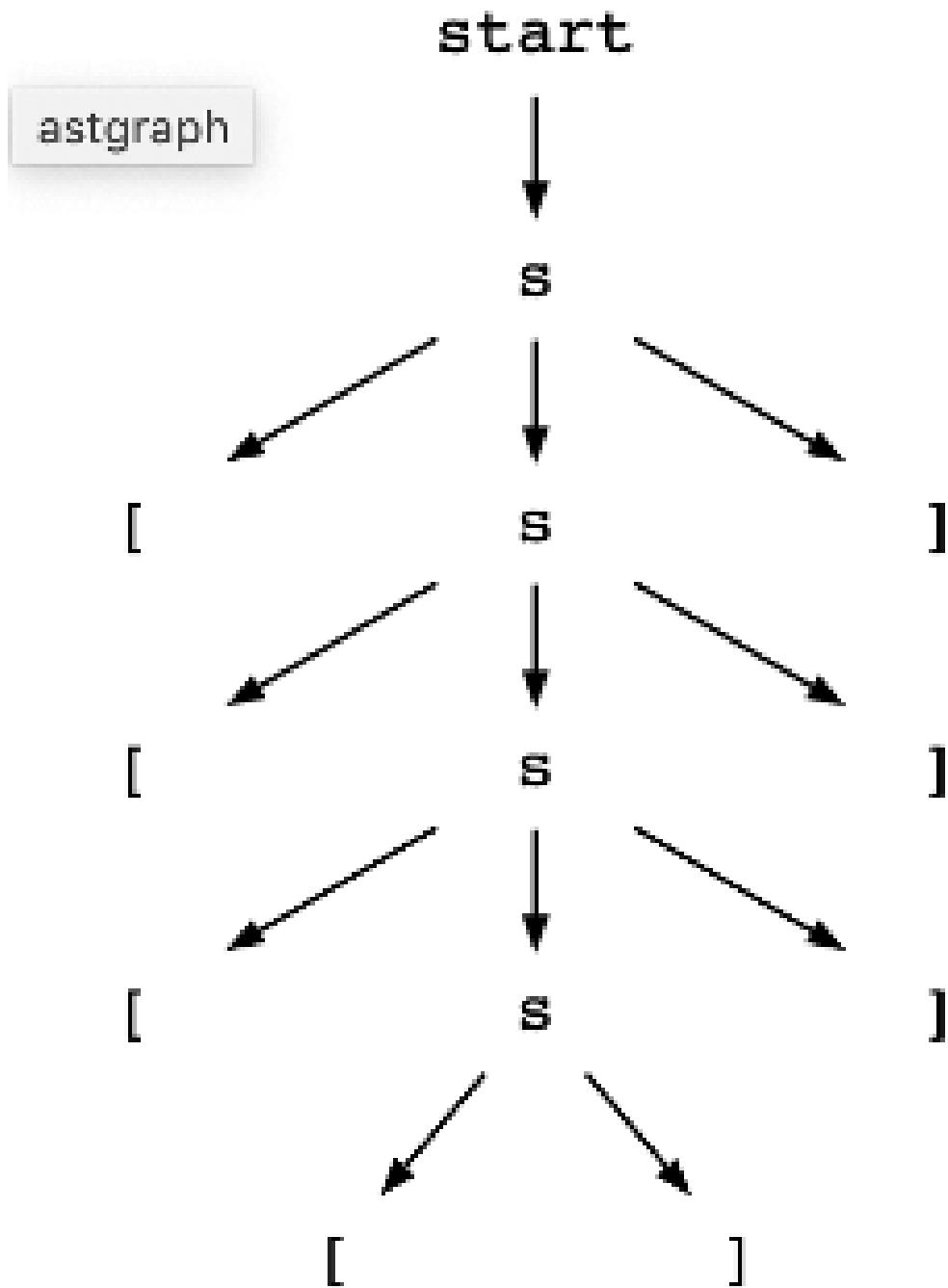
Stelsels van dit soort van namen voorziene expressies die recursief naar deelexpressies kunnen verwijzen noemen we **grammatica's**. Het is de standaardmanier om de structuur van formele talen zoals programmeertalen te beschrijven.

Bij een gegeven grammatica kunnen we **parse trees** (parseerbomen) maken die laten zien welke regels zijn toegepast. Dit zijn boomvormige structuren waarin voor iedere naam die voorkomt in een regel er onder is aangegeven welke alternatief gekozen is. Als het voor een bepaalde string lukt om een bijbehorende parse tree te maken weet je dat die string onderdeel is van de taal.

Neem bijvoorbeeld weer een simpele grammatica met slechts één (recursieve) regel:

$S = '[' ']' \mid '[' S ']'$

De string [[[[]]]] kunnen we dan ontleden tot de volgende boom:



### 3.9 Rekenkundige expressies

De taal van de rekenkundige expressies bestaat uit strings zoals:

182

$$13 * 17 - 5 + 18 \\ (1 + 8) * (7 - 14)$$

Voor sommen zonder haakjes zouden we met de volgende simpele grammatica de sommen kunnen beschrijven:

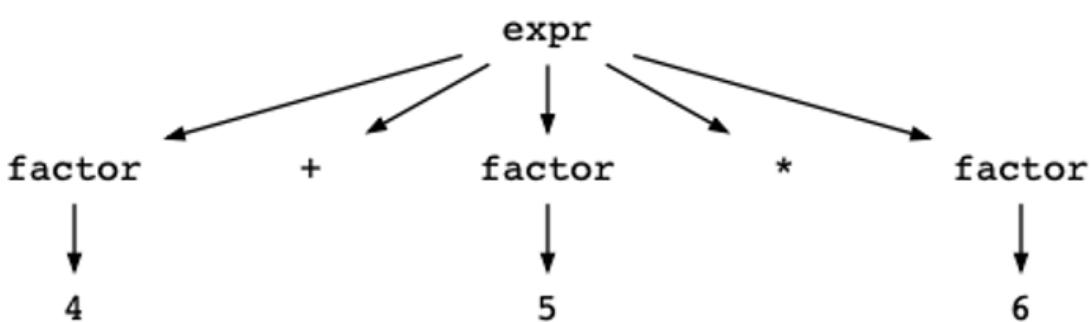
```
expressie = getal (oper getal)*
oper      = '+' | '*' | '/' | '-'
```

In dit geval hebben we nog genoeg aan de uitdrukkingskracht van de reguliere expressies. We kunnen oper immers gewoon invullen in expressie.

Als we ook sommen met haakjes willen beschrijven moeten we een (indirect) recursieve regel invoeren:

```
expressie = factor (oper factor)*
oper      = '+' | '*' | '/' | '-'
factor    = getal | '(' expressie ')'
```

We kunnen hiermee bijvoorbeeld de expressie  $4 + 5 * 6$  ontleden:



Als ons enige doel het herkennen van correcte rekenkundige expressies was, was deze grammatica voldoende. Als we de expressies ook uit willen rekenen is het handig om onderscheid te maken tussen de prioriteiten van de verschillende operatoren.

Dit kun je in een grammatica eenvoudig doen door een extra regel tussen te voegen:

```
expressie = term   (plusoper term)*
term      = factor (multoper factor)*
plusoper = '+' | '-'
multoper = '*' | '/'
factor    = getal | '(' expressie ')'
```

### 3.10 Turing Machines

Turing machines zijn het meest krachtige model van berekening dat we in deze cursus bespreken. Je kunt ze beschouwen in de context van een aantal fundamentele vragen:

- Zijn de begrippen algoritme en computer te formaliseren? (en wat is het verband tussen beide?)
- Welke computermodellen zijn mogelijk? (en is er verschil in rekenkracht?)
- Zijn er beperkingen aan wat er überhaupt met een computer berekend kan worden?

De modellen die we tot nu gezien hebben zijn nuttig maar kunnen niet alle talen beschrijven.

**Eindige automaten** kun je zien als primitieve computers, maar de taal  $\{ a^i b^i \mid i \geq 0 \}$  kunnen we er niet mee beschrijven.

**Pushdown automaten** kunnen meer dan eindige automaten, maar de taal  $\{ a^i b^i c^i \mid i \geq 0 \}$  valt buiten de mogelijke talen.

### 3.10.1 Wat zijn Turing machines?

Turing bedacht zo rond 1935 (voordat er echte computers waren) een computer model gebaseerd op dat van de eindige automaat. Het blijkt dat dit model universeel is.

- ieder ander computer model dat je kunt verzinnen is niet krachtiger dan een Turing machine.
- ieder algoritme dat met een willekeurige computer kan worden uitgevoerd kan ook met een Turing machine worden uitgevoerd.

Een (informele) definitie van Turingmachines is als volgt:

- Het zijn eindige automaten met als extra een oneindig grote "tape" met symbolen waarop gelezen en geschreven kan worden.
- Er is een "leeskop" die op bepaalde positie van tape staat.
- In iedere state wordt de volgende state bepaald door de combinatie van de state en de waarde onder de leeskop.
- Bij een overgang kan een ander symbool op de plek van de leeskop worden geschreven.
- Bij een overgang beweegt de leeskop een positie naar links of naar rechts.
- Er zijn speciale eindtoestanden (states) waar de berekening stopt.
  - Dit kan een "Accept" of "Reject" state zijn
  - Vaak is de inhoud van de tape na afloop het "antwoord"
- Vaak ontbreken de "Reject" states en is de "Reject" impliciet als er geen regel toegepast kan worden

**Voorbeeld**  $\{whwb \mid w \in (0|1)^*\}$  (bijv. 110h110b of 11011h11011b).

Deze machine controleert of twee strings bestaande uit enen en nullen hetzelfde zijn. De 'h' en 'b' symbolen markeren het einde van de twee strings.

Tijdens het controleren beweegt de leeskop steeds tussen het stuk voor de 'h' en het stuk na de 'h'. Voor iedere '1' in het eerste deel wordt een bijbehorende '1' in het tweede deel gezocht. Voor een '0' in eerste deel moet juist een bijbehorende '0' in het tweede deel aanwezig zijn. De symbolen '0' en '1' worden tijdens de berekening vervangen door het symbool 'x' om de voortgang bij te houden.

Overgangen:

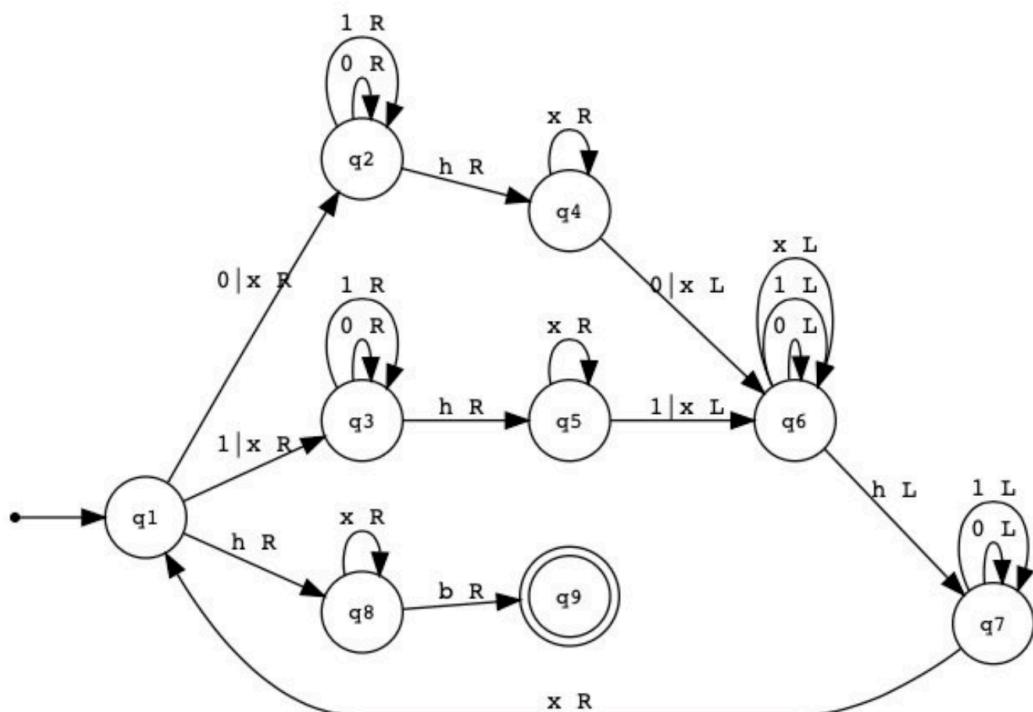
```

q1 0 x R q2
q1 1 x R q3
q2 0 0 R q2
q2 1 1 R q2
q2 h h R q4
q3 0 0 R q3
q3 1 1 R q3
q3 h h R q5
q4 x x R q4
q4 0 x L q6
q5 x x R q5
q5 1 x L q6
q6 h h L q7
  
```

```

q6 0 0 L q6
q6 1 1 L q6
q6 x x L q6
q7 0 0 L q7
q7 1 1 L q7
q7 x x R q1
q1 h h R q8
q8 x x R q8
q8 b b R q9
accept q9
start q1

```



**Voorbeeld**  $\{0 z^n b \mid n \geq 0\}$  ( $2^n$  nullen, spatie op eind, oob oooob oooooooob)

Deze machine kijkt dus feitelijk of aantal symbolen een macht van  $z$  vormt. Dit doet de machine door meerdere keren over de input string te gaan en steeds de helft van de nullen die hij tegen komt te vervangen door een 'x'. Voor machten van  $z$  kun je dit een exact aantal keer doen, voor andere getallen kom je verkeerd uit.

Overgangen:

```

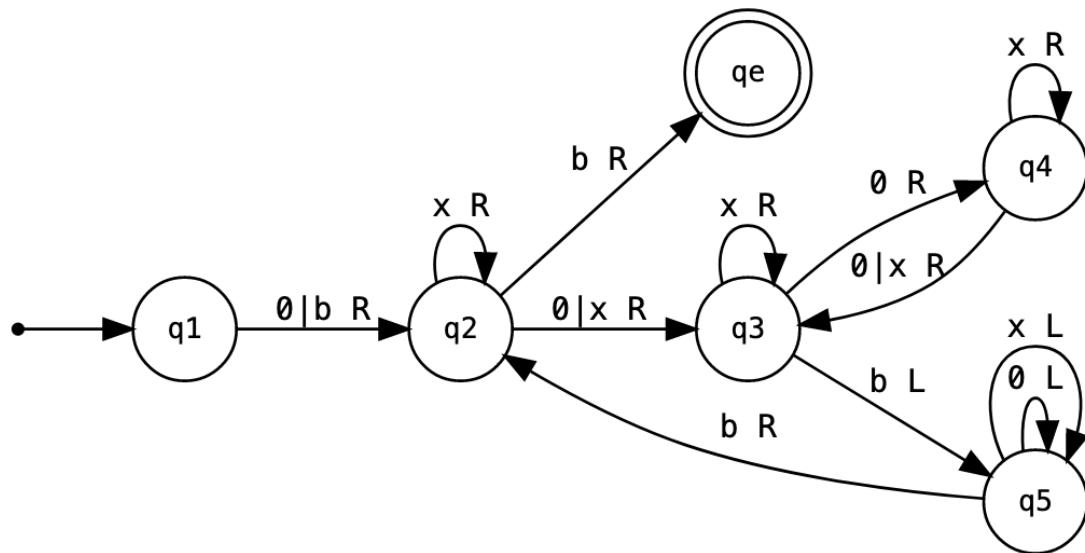
q1 0 b R q2
q2 b b R qe
q2 x x R q2
q2 0 x R q3
q3 x x R q3
q3 0 0 R q4
q3 b b L q5
q4 x x R q4
q4 0 x R q3
q5 0 0 L q5
q5 x x L q5

```

```

q5 b b R q2
start q1
accept qe

```



Het gedrag van een Turing machine is eenvoudig te simuleren in iedere programmeertaal. In Python is het volgende programma bijvoorbeeld al een volledige Turing machine:

```

def sim_turing(tm,tape):
    ts,start,ess,rss = tm
    state = start
    pos = 0
    while state not in ess and state not in rss:
        sym = tape[pos]
        res = [(ws,d,qe) for q,rs,ws,d,qe in ts if state == q and sym == rs]
        if len(res) == 0:
            # print("no rule for: ",state,sym)
            return ('Reject',tape)
        else:
            # print(res[0])
            tape[pos],d,state = res[0]
            pos += 1 if d == 'R' else -1
    if state in ess:
        return ('Accept',tape)
    else:
        return ('Reject',tape)

```

Het simuleren van Python met een Turing machine is uiteraard een stuk lastiger, maar niet onmogelijk.

### 3.10.2 Varianten

Naast de standaard Turing machine bestaan er ook allerlei varianten van Turing machines.

**Multi-track Turing machines:**

Dit is een Turing machine met meerdere tapes met ieder een eigen leeskop (gelijktijdig lezen en schrijven). Iedere multi-track TM is met een standaard TM te simuleren.

#### **Multi-tape Turing machines:**

Dit is vergelijkbaar met een multi-track Turing machine, maar met de mogelijkheid om de leeskoppen los van elkaar te bewegen.

**Non-deterministische Turing machines:** Net als bij de eindige automaten zijn non-deterministische Turing machines (NTM) hetzelfde als gewone Turing machines (DTM) maar is de volgende toestand (en geschreven symbool, etc) niet uniek bepaald door een combinatie van toestand en symbool. Voor iedere NTM is er een equivalente DTM (net als bij eindige automaten). Dit gaat dmv simulatie mbv een multi-tape machine, maar valt buiten de scope van deze cursus.

**Universele Turing machines:** Dit is een Turing machine waarmee we iedere andere Turing machine kunnen simuleren.

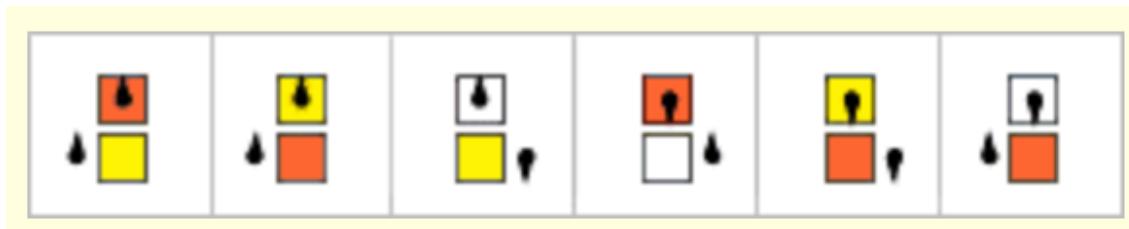
Dit zou bijvoorbeeld een Turing Machine met 2 tapes kunnen zijn:

- Een tape bevat een “programma” die een andere Turing machine beschrijft
- Een andere tape bevat de invoer code voor dit programma

Dit is dus in feite een gewone computer. Daar staat het programma ook in het geheugen.

De kleinste universele Turing machine (<http://www.wolframscience.com/prizes/tm23/>) heeft twee toestanden (up en down) en 3 symbolen (kleuren: white red en yellow).

Het volgende plaatje geeft de overgangen weer:



```

u r y L u
u y r L u
u w y R d
d r w R u
d y r R d
d w r L u

```

Ieder berekenbaar probleem kan dus naar een input tape bestaande uit w,ry omgezet worden, waarna de oplossing door deze TN berekend wordt!

### 3.11 Het Halting Probleem

Een belangrijk resultaat uit de informatica is dat niet alle problemen beslisbaar zijn. Dat wil zeggen, dat er voor een probleem een algoritme bestaat dat altijd een antwoord geeft en niet oneindig door blijft rekenen.

Het zogenaamde “Halting probleem” is een uitspraak over Turing machines die dit laat zien. Het is als volgt gedefinieerd:

Bestaat er een programma (TM of programma voor universele TM) **H** dat gegeven een willekeurig programma **P** met input **I** voor dit programma aangeeft of dit programma stopt.

Als dit programma **H** zou bestaan, dan kunnen we een groot aantal open wiskundeproblemen makkelijk oplossen.

Neem bijvoorbeeld het Goldbach vermoeden:

Ieder even getal groter dan 2 is te schrijven als de som van 2 priemgetallen.

Dit is een beroemd (nog) onbewezen vermoeden!

We kunnen een eenvoudig programma schrijven dat voor ieder getal (groter dan 2) naar een oplossing zoekt.

```
def is_prime(n):
    wn = math.sqrt(n)
    wn = math.trunc(wn)
    for k in range(2,wn+1):
        if n % k == 0:
            return False
    return True

def is_sum_of_2_primes(n):
    for k in range(2, n // 2+1):
        if is_prime(k) and is_prime(n-k):
            return True
    return False

def goldbach_bound(b):
    n = 4
    while n < b:
        if not is_sum_of_2_primes(n):
            return False
        n += 2
    return True

def goldbach():
    n = 4
    while True:
        if not is_sum_of_2_primes(n):
            return False
        n += 2
    return True
```

Een aanroep van de begrensde versie, `goldbach_bound(b)`, kan voor een willekeurige bovengrens bepalen of alle even getallen kleiner dan de bovengrens voldoen.

Een aanroep van `goldbach()` stopt echter alleen<sup>1</sup> als er een even getal is, dat niet de som is van priemgetallen.

Als het programma **H** zou bestaan konden we bepalen of `goldbach()` wel of niet stopt en kunnen we dus het Goldbach vermoeden bewijzen/ontkrachten.

### 3.11.1 Paradoxen

Om te bewijzen dat **H** niet bestaat creëren we een paradox vergelijkbaar met de volgende uitspraken:

- “Deze zin is niet waar!”

---

<sup>1</sup>Als we aannemen dat we oneindig veel tijd en geheugen hebben

- Iemand is ter dood veroordeeld, maar mag zelf kiezen of hij wordt onthoofd of opgehangen. Hij mag een uitspraak doen. Als de uitspraak waar is wordt hij onthoofd. Als hij niet waar is wordt hij opgehangen. Waarop hij zegt: "Ik word opgehangen!" waardoor het onmogelijk is de straf uit te voeren. (Als hij wordt opgehangen had hij onthoofd moeten worden)

### 3.11.2 Notatie

We introduceren de volgende notatie:

- Een programma geven we aan met een hoofdletter: **H**, of **P**.
- De representatie van een programma **P** duiden we aan met **<P>** Dit is bijvoorbeeld een unicode/ascii string of een reeks nullen en enen. We nemen aan dat de input met dezelfde symbolen wordt gerepresenteerd (dit is essentieel voor het bewijs).
- Het uitvoeren van programma **P** met inputs **I** en **J** noteren we als **P I J**.

Het Halting probleem is dan te formuleren als:

Bestaat er een programma **H** zodanig dat:

1. **H <P> I = True**, als **P I** termineert
2. **H <P> I = False**, als **P I** niet termineert

### 3.11.3 Bewijs

We formuleren een bewijs uit het ongerijmde: we nemen dus aan dat **H** bestaat en kijken of dat tot een tegenspraak leidt. We definieren het programma **D** dat net als **H** een programma als input heeft:

1. **D <P> = loop()**, als **H <P> <P> = True**
2. **D <P> = True**, als **H <P> <P> = False**

De functie **loop()** is hierbij een oneindige lus.

We bekijken dus het geval waarin het programma **P** wordt toegepast op zijn eigen codering **<P>**. Dit mogen we doen omdat de invoer en programmacodering dezelfde symbolen gebruiken. Wat dit betekent vragen we ons niet af.

We kunnen daarom ook het geval bekijken waarin **D** wordt toegepast op zijn eigen representatie **<D>**: **D <D>**.

Stel **D <D>** termineert dan geldt er:

- **H <D> <D> = False** volgens de definitie van **D** en dus:
- **D <D>** termineert niet volgens de definitie van **H**: tegenspraak!

Stel dat **D <D>** niet termineert dan:

- **H <D> <D> = True** volgens de definitie van **D** en dus:
- **D <D>** termineert volgens de definitie van **H**: tegenspraak!

In beide gevallen hebben we dus een tegenspraak. Kortom **H** kan niet bestaan!

### 3.11.4 Meer onbeslisbare problemen

Een ander voorbeeld van een ogenschijnlijk eenvoudig, maar onbeslisbaar probleem, is het "Post Correspondence Probleem":

- Stel we hebben een verzameling “dominosteentjes” met op de boven- en onderkant een (verschillende) “string”.
- Is het mogelijk deze steentjes zo op volgorde te leggen dat de reeks onderkanten en de reeks bovenkanten dezelfde string vormen. Hierbij mogen steentjes meerdere keren gebruikt worden.

Op Wikipedia ([https://en.wikipedia.org/wiki/Post\\_correspondence\\_problem](https://en.wikipedia.org/wiki/Post_correspondence_problem)) is een bewijs geschetst waarom het niet mogelijk is om een algoritme te verzinnen dat voor iedere verzameling steentjes kan aangeven of er oplossing voor bestaat.

In het bewijs laat men zien dat het simuleren van een willekeurige Turingmachine gecodeerd kan worden met behulp van deze dominosteentjes. Het vinden van de correspondentie is dan hetzelfde als de oplossing van het Halting probleem. Aangezien dat niet mogelijk is, is het ook niet mogelijk om het Post probleem in het algemeen op te lossen!

Een ander bekend probleem dat niet beslisbaar is Hilberts 10e probleem: Een stelling over zogenaamde “Diophantische vergelijkingen”.

Beschouw vergelijkingen van het type:

$$a \times x^n + b \times y^m + c \times z^k = 0$$

Waarbij met  $a, b, c, n, m, k$ , gehele (mogelijk negatieve) getallen zijn.

Neem bijvoorbeeld:  $x^3 + y^3 + z^3 = 42$

Bestaat er een oplossing van deze vergelijking met gehele getallen  $x, y, z$ ?

Dit is nog onbekend voor  $q2$  (geldt wel voor andere  $n < 100$ ). Dit probleem is ook in het algemeen onbeslisbaar. Er kan geen methode bestaan die voor een willekeurige Diophantische vergelijking kan zeggen of deze een oplossing heeft (Negatief antwoord op Hilberts 10e probleem).

## 3.12 Talen en Processoren

Turing machines kunnen in principe alles uitrekenen wat berekenbaar is, maar als we ieder algoritme als een Turing machine zouden moeten beschrijven zou dat wel erg omslachtig zijn. Kortom, een Turing machine is niet heel praktisch voor echte (programmeer-) problemen>

Voor praktisch werkbare talen om algoritmes mee te beschrijven (programmeertalen) hebben we aanvullende eisen:

- Een programmeur moet zonder al te veel moeite kunnen programmeren: algoritmen zijn makkelijk uit te drukken
- De taal moet efficient door een moderne computer verwerkt kunnen worden

Als we kijken naar een subset van een taal als Python zien we verschillende concepten die het uitdrukken van algoritmes makkelijker maken. Beschouw bijvoorbeeld het volgende programma dat een faculteit uitrekent:

```
def fac(n):
    res = 1
    k = 1
    while k <= n:
        res = res * k
        k = k + 1
    return res

print(fac(5))
```

We zien hier al veel concepten:

- `def fac(n):` Definitie van functies
- `k` en `res`: Variabelen om geheugenlocaties te benoemen
- `k = 1` en `res = 1`: Toekenningen van geheugen
- `while k <= n:` Conditioneel uitvoeren van delen van een programma
- `res = res * k` is het berekenen van een expressie en een toekenning
- `return res` Terug geven van resultaten van functies

Voor al deze concepten moeten implementaties van programmeertalen vertalingen kunnen maken naar de instructieset van een (moderne) processor.

Een moderne microprocessor benaderd de mogelijkheden van een Turing machine maar is toch net iets anders van opzet:

- Hij voert simpele operaties die (in principe) sequentieel worden uitgevoerd
- De instructies staan in het geheugen
- De instructie die wordt uitgevoerd staat op positie `pc` (program counter)
- Na iedere instructie wordt `pc` met 1 verhoogd
- Instructies kunnen waarden in geheugen gebruiken en wijzigen

Daarnaast is er natuurlijk geen oneindig lange tape maar een werkgeheugen: Dit is een aaneengesloten reeks geheugencellen die een programma kan lezen en schrijven. Je kunt het zien als een groot array, maar wordt logisch opgedeeld in een aantal delen:

- **Programma gedeelte:** waar de reeks van instructies staat
- **Stack gedeelte:** waar lokale variabelen functies, functie administratie en tijdelijke waarden in berekening staan
- **Heap gedeelte:** waar grotere datastructuren en globale variabelen staan
- **(Constant pool gedeelte):** waar strings en andere constante waarden uit het programma staan

## 3.13 Virtuele machines

De instructieset van een bepaald soort processor is ook weer een taal. Deze taal wordt direct “begrepen” door de hardware. De processor hardware is eigenlijk een interpreter voor de instructies, en de instructies bepalen wat je processor kan laten doen.

Wanneer verschillende processoren dezelfde instructieset (taal) ondersteunen zijn ze compatibel. Zo kun je een programma bedoeld voor een Intel i86 processor ook uitvoeren met een AMD processor. Dat wil overigens niet zeggen dat de hardware identiek is, deze kan heel verschillend zijn (bijvoorbeeld sneller of efficienter).

### 3.13.1 Virtuele Machines

Als we een instructieset zien als een taal die de mogelijkheden van een processor definieert, kunnen we ook processoren verzinnen die niet in hardware bestaan. Dit noemen we virtuele machines (met virtuele instructies).

Om een programma voor zo’n virtuele machine uit te voeren gebruik je een interpreter die je uitvoert op een andere (fysieke) processor. Een virtuele machine is meestal eenvoudiger dan een in hardware uitgevoerde processor. De instructies voor een virtuele machine noemen we **byte code** en veel programmeertalen vertalen programma’s als eerste stap naar bytecode voor een virtuele machine. Bijvoorbeeld:

- Java Virtuele Machine (JVM)
- Python virtuele machine (CPython)
- WebAssembly voor snelle executie in webbrowsers

In deze cursus zullen in meer detail kijken naar een vereenvoudigde versie van de JVM. Deze is eenvoudiger dan de virtuele machine van bijvoorbeeld Python.

### 3.13.2 Java Virtuele Machine JVM

Om een Java of Python (Jython) programma uit te kunnen voeren wordt het eerst vertaald naar byte code. Deze byte code instructies worden uitgevoerd door de JVM.

Hoewel de byte code in principe in hardware gerealiseerd zou kunnen worden wordt hij normaalgesproken in software uitgevoerd door een interpreter. Om dichter bij de performance van in hardware uitgevoerde instructies te komen bevat deze interpreter een Just-In-Time (JIT) compiler. Deze vertaalt de bytecode vlak voor het uitvoeren naar instructies voor de concrete processor waar de JVM op draait.

De JVM heeft een zogenaamde “stack architectuur”. Dat wil zeggen dat de instructies uitgaan van de aanwezigheid van een stack. Rekenkundige operaties werken op de stack, lokale variabelen staan op de stack, en argumenten en return waarden van functies staan ook op de stack. Het vertaalde programma (de instructies) staan apart van de stack in het geheugen van de JVM. Ze worden na elkaar uitgevoerd, maar er zijn ook instructies om naar een andere instructie te springen.

De stack van de JVM is vergelijkbaar met die in een pushdown automaat. Je kunt bovenaan waardes toevoegen en verwijderen (push en pop). Rekenkundige bewerkingen zijn altijd van toepassing op de bovenste (1 of 2) elementen. Wat anders is dat de stack ook wordt gebruikt om lokale variabelen en functie argumenten in op te slaan. Die zijn dan toegankelijk binnen een gehele functieaanroep. Je kunt dus gedeeltelijk bij elementen die niet helemaal bovenaan de stack staan.

De instructies van de JVM zijn grofweg in drie categorieën in te delen:

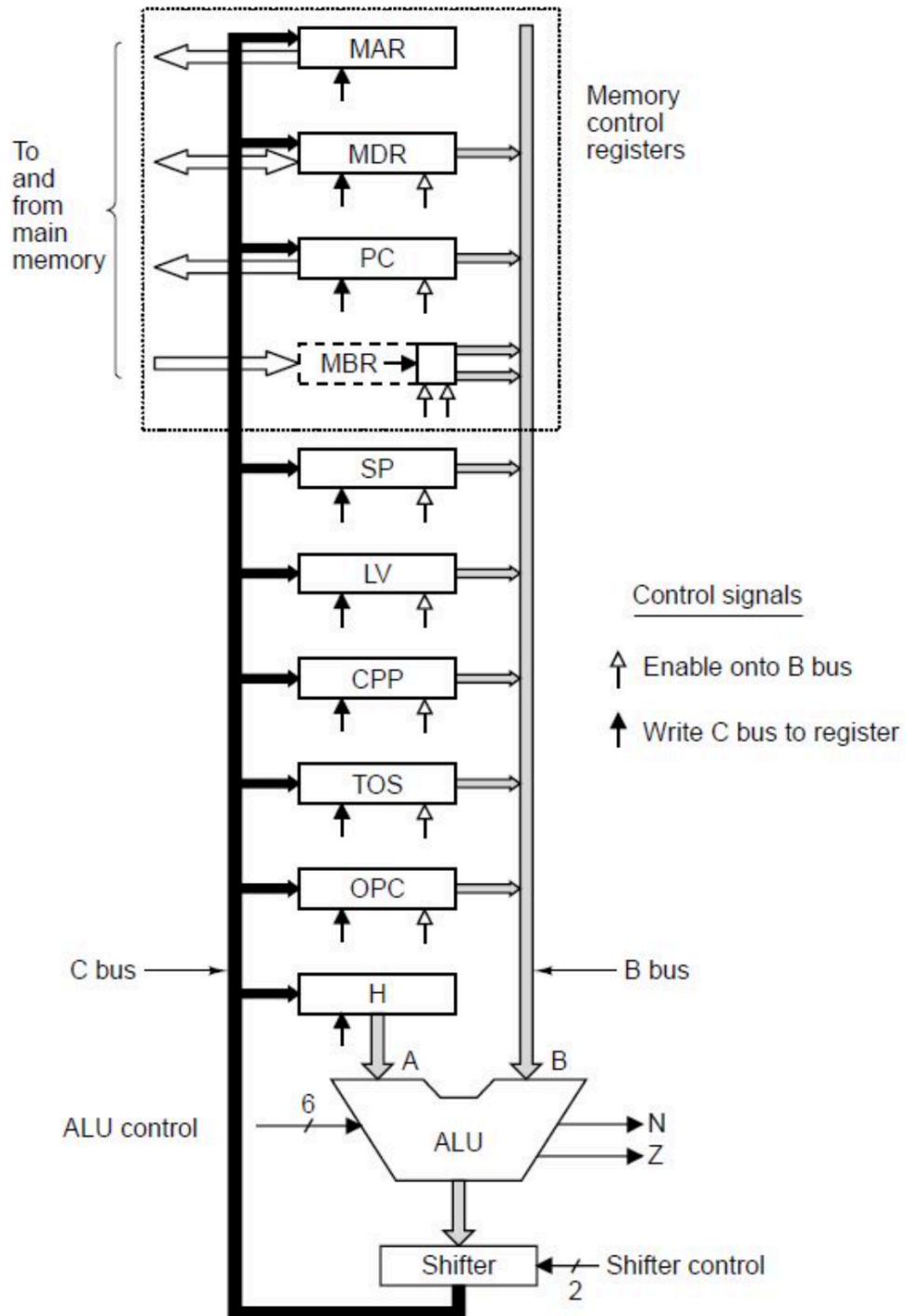
- Rekenkundige instructies
  - Optellen, aftrekken, vermenigvuldigen etc.
  - De operanden staan op de stack, en het resultaat komt ook op de stack
- Branching (sprong) instructies
  - Springen naar instructies (goto)
  - Springen met een voorwaarde (if, while)
  - Springen naar een functie
- Geheugen instructies
  - Waardes laden vanuit een geheugenlocatie naar (de top van) de stack
  - Waardes opslaan naar een geheugenlocatie vanaf (de top van) de stack

Voor de subset die we in detail bekijken (IJVM) behandelen we voor ieder van de instructies hun precieze werking.

## 3.14 De IJVM Machine

De IJVM instructieset is een versimpelde versie van de JVM instructieset die werkt voor programma's die enkel rekenen met integers (of arrays van integers), en niet met andere types objecten. Deze subset is krachtig genoeg om de basisconstructies van talen als Java of Python op af te beelden. Hij is gebaseerd op de virtuele machine uit het boek “Structured Computer Organization” van Tanenbaum en Austin.

In de voorbeelden zullen we Python gebruiken om de vertaling van de abstractere programmeertaalconcepten naar IJVM instructies te illustreren.



De bovenstaande afbeelding toont het datapad van de IJVM virtuele machine. De blokken tonen de registers (de vaste geheugens in de processor), de rekeneenheid (ALU) en hoe deze met elkaar en met het werkgeheugen verbonden zijn.

De relevante registers zijn:

- **SP:** Hierin staat het adres van de top van de stack
- **PC:** Hierin staat het adres van de volgende instructie
- **LV:** Hierin staat de locatie (op de stack) van de argumenten en lokale variabelen van de huidige functie.
- **TOS:** Hierin staat een kopie van het bovenste element van de stack

De onderstaande tabel geeft een overzicht van de instructies van de IJVM machine:

---

bipush const	zet constante op stack
goto offset	zet pc op pc + offset
iadd	tel bovenste 2 elementen van stack op en zet resultaat op stack
isub	trek bovenste 2 elementen van stack af ...
imult	vermenigvuldig bovenste 2 elementen van stack ...
idiv	deel bovenste 2 elementen van stack ...
iand	boolean and bovenste 2 elementen van stack ...
ior	boolean or bovenste 2 elementen van stack ...
iinc vn const	verhoog variabele met offset vn tot LV met const
iload vn	laad variable met offset vn tot LV op stack
istore vn	pop waarde van stack en zet dit in variabele met offset vn tot LV
dup	dupliceer top van stack
pop	pop waarde van stack
swap	verwissel bovenste 2 elementen stack
iflt offset	pop bovenste waarde stack, als < o ga verder bij pc + offset
ifeq offset	pop bovenste waarde stack, als = o ga verder bij pc + offset
if_compeq offset	pop bovenste 2 waarden stack, als gelijk ga verder bij pc + offset
call nrargs nrlv npc	roep functie aan
ireturn	return uit functie

---

### Rekenkundige instructies

De rekenkundige operaties werken op de bovenste elementen van de stack. Operanden moet je dus voor de operatie op de stack pushen. Bijvoorbeeld:

```
bipush 6
bipush 7
iadd
```

Tijdens de iadd worden de 6 en 7 van de stack gepopt en wordt het resultaat er op gepusht. Na afloop staat er dus 13 boven op de stack.

Door deze volgorde kun je ook ingewikkelde expressie uitrekenen in een specifieke volgorde. Neem bijvoorbeeld  $3 + 6 * 7$ . Volgens de rekenregels moet eerst de vermenigvuldiging gebeuren. Dit kunnen we eenvoudig voor elkaar krijgen:

```
bipush 3
bipust 6
bipush 7
imult
iadd
```

Eerst wordt  $6 * 7$  uitgerekend, en het resultaat ervan wordt opgeteld bij de 3 die ongewijzigd op de stack was blijven staan tijdens de vermenigvuldiging.

Als we daarentegen  $(3 + 6) * 7$  willen uitrekenen kunnen we simpelweg de iadd eerder uitvoeren:

```
bipush 3
bipust 6
iadd
bipush 7
imult
```

Andersom kun je natuurlijk ook de expressie achterhalen bij een gegeven reeks instructies:

```
bipush 4
bipush 5
bipush 8
bipush 6
isub
imult
iadd
```

Hier worden eerst 8 en 6 opgeteld, vervolgens wordt 5 met het resultaat er van vermenigvuldigd, en als laatste wordt dit bij 4 opgeteld (dus  $4 + 5 * (8 + 6)$ ).

### Branching

De branching instructies `goto`, `if_lt`, `if_eq` en `if_compeq` gebruik je om de program counter (PC) te verzetten en zo te springen in het programma. Deze instructies krijgen allemaal als vast argument een offset mee. Dit is de instructie waar naar gesprongen moet worden. Dit is het aantal stappen dat je vooruit of terug moet springen.

Omdat het handmatig tellen van het aantal instructies dat je vooruit moet springen vervelend en foutgevoelig is gebruiken we in de voorbeelden in plaats van offsets labels. Er staat dan een label voor een bepaalde instructie en op de plaats waar we een offset nodig hebben schrijven we het label.

### Functies

De `call` en `ireturn` instructies zijn de meest complexe in de instructieset. Samen stellen ze je in staat om functies aan te roepen. De ingredienten van een functie zijn:

- Een functienaam om de functie te identificeren
- Een aantal argumenten
- Een aantal lokale variabelen
- Een functiebody waarin iets gedaan wordt met de lokale variabelen
- Een returnwaarde

Omdat de JVM een stack gebaseerde machine is gebruiken we de stack om argumenten door te geven en om het resultaat te returnen.

Concreet bestaat een functiecall uit drie stappen:

1. Zorg dat alle benodigde argumenten op de stack staan
2. Voer de `call` instructie uit
3. Doe iets met het resultaat.

Na de `call` instructie zijn de argumenten van de stack afgehaald en staat de returnwaarde op de top van de stack.

De `call` instructie heeft een aantal parameters:

- `nrargs` is het aantal argumenten dat de functie heeft.
- `nrvl` is het aantal lokale variabelen dat de functie gebruikt
- `npc` is de offset/label van de instructies die de functiebody beschrijven

Tijdens de `call` instructie:

1. Staan de argumenten van de functie al klaar op de stack
2. Worden `nrvl` extra plaatsen op de stack gereserveerd voor lokale variabelen
3. Worden de huidige PC en LV adressen hier boven op gezet
4. Krijgt LV het adres van het eerste argument (een geheugenadres in de stackruimte)
5. Krijgt PC het adres van `npc` waardoor je naar de functiebody springt

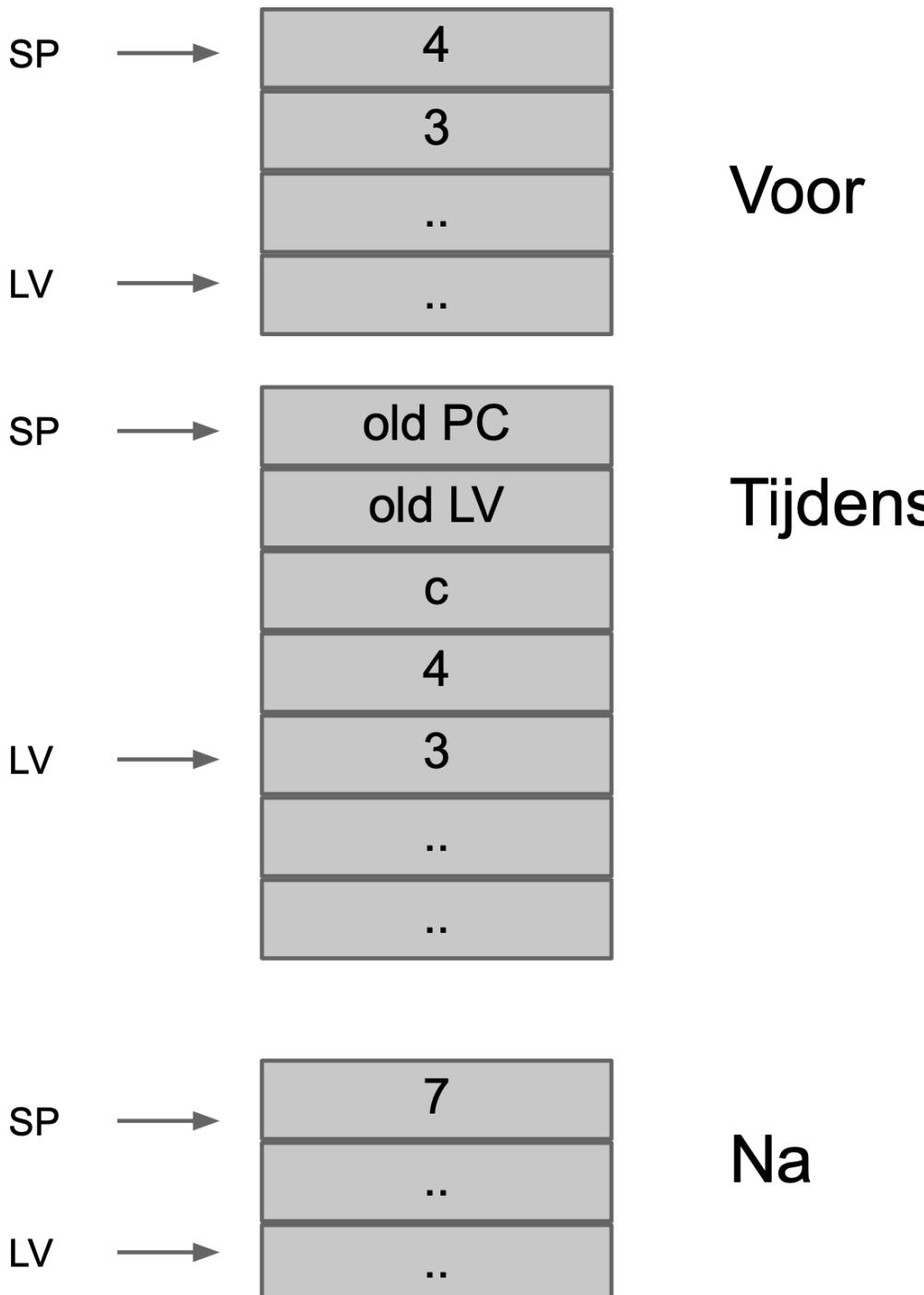
Neem bijvoorbeeld de hele simpele functie f:

```
def f(a, b):
    c = a + b
    return c

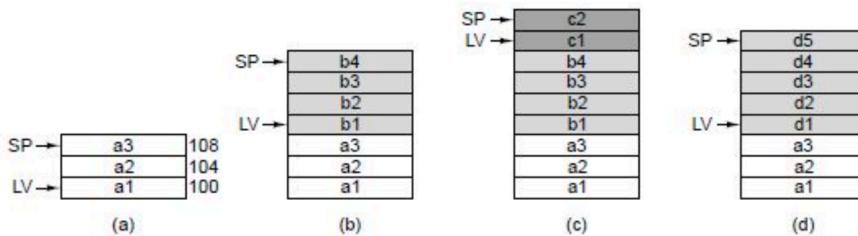
x = f(3, 4)
```

De aanroep ziet er dan als volgt uit:

```
bipush 3
bipush 4
call 2 1 label
```



De `call` instructie zorgt er voor dat bij iedere functie aanroep een apart segment van de stack wordt gereserveerd. Dit noemen we een **stack frame**. In dat segment staan de argumenten en lokale variabelen waardoor meerdere functie aanroepen van dezelfde functie hun eigen lokale variabelen hebben (kan zelfs recursief). Daarnaast zorgt het bewaren van de huidige LV en PC registers op de stack er voor dat de `ireturn` call het gehele stack frame weer kan opruimen en de processor weer verder kan gaan waar hij gebleven ten tijde van de `call` instructie.



**Figure 4-8.** Use of a stack for storing local variables. (a) While *A* is active. (b) After *A* calls *B*. (c) After *B* calls *C*. (d) After *C* and *B* return and *A* calls *D*.

Deze opzet van functies aanroepen met stack frames per functie is niet de enige manier om functies te realiseren. De eerste generaties computer en programmeertalen maakten geen gebruik van een stack voor argumenten en lokale variabelen. In plaats daarvan lagen alle posities van lokale variabelen van te voren vast. Hierdoor waren recursieve functies niet mogelijk omdat een nieuwe aanroep van een functie voordat deze klaar was de argumenten overschreef.

**Voorbeelden** Om de IJVM machine te illustreren geven we nog een aantal voorbeelden in Python met een equivalent in IJVM instructies. De voorbeelden bevatten zowel een functiedefinitie als een functie aanroep.

De extra instructies `print`, `printstack` en `stop` zijn geen onderdeel van de IJVM instructieset, maar zijn bedoeld voor de IJVM simulator (zie uitleg verderop). Ze laten respectievelijk de top van de stack zien, laten de gehele stack zien, of stoppen het programma.

De volgende Python code:

```
def telop(a,b):
    c = a + b
    return c

print(telop(3,4))
```

Is in IJVM:

```
bipush 3
bipush 4
call 2 1 33
print
stop
33 iload 0
iload 1
iadd
istore 2
iload 2
ireturn
```

Deze while loop in Python:

```
def sumloop(n):
    s = 0
    while n != 0:
        s = s + n
```

```

n = n- 1
return s

sumloop(10)

```

Is in IJVM:

```

bipush 10
call 1 1 1
print
stop
1 bipush 0
istore 1
2 printstack
iload 0
ifeq 3
iload 1
iload 0
iadd
istore 1
iload 0
bipush 1
isub
istore 0
goto 2
3 iload 1
ireturn

```

Een recursieve variant in Python:

```

def sumrec(n):
    if n == 0:
        return 0
    else:
        return n + sumrec(n-1)

sumrec(10)

```

Wordt in IJVM instructies:

```

bipush 10
call 1 0 1
printstack
print
stop
1 printstack
iload 0
ifeq 2
iload 0
dup
bipush 1
isub
call 1 0 1
iadd
printstack

```

```

        ireturn
2   bipush 0
    ireturn

```

En het recursief oplossen van de ‘Torens van Hanoi’ in Python

```

def hanoi(n,a,b,c):
    if n > 0:
        hanoi(n-1,a,c,b)
        print(a,b)
        hanoi(n-1,c,b,a)

hanoi(3,1,2,3)

```

Is in IJVM instructies:

```

bipush 3
bipush 1
bipush 2
bipush 3
call 4 0 33
pop
stop
33 iload 0
ifeq 44
iload 0
bipush 1
isub
iload 1
iload 3
iload 2
call 4 0 33
pop
iload 1
print
iload 2
print
iload 0
bipush 1
isub
iload 3
iload 2
iload 1
call 4 0 33
pop
bipush 77
ireturn
44 bipush 66
ireturn

```

### **Simulator**

Om te experimenteren met IJVM programma's is er een online simulator beschikbaar:

<https://tvit24.cloud.cl2ab.nl/IJVM/>

De operaties die je met de simulator kunt uitvoeren zijn:

<b>Run</b>	voer het programma uit tot het eind
<b>Load</b>	(her)laad het programma voor step
<b>Step</b>	voer 1 instructie uit
<b>Reset</b>	ga in step weer terug naar instructie 1 en wis het uitvoerscherm
<b>Clear</b>	wis het uitvoerscherm
<b>Stop</b>	onderbreek een programma dat in een oneindige loop is geraakt

## 3.15 Heap geheugen

Het geheugen dat gebruikt wordt voor de stack is afdoende voor lokale variabelen en functieargumenten. Vanwege het stack gedrag is het na iedere return weer beschikbaar voor de volgende functiecall en bestaan lokale variabelen alleen tijdens de uitvoer van de functie.

Toch heb je in programma's ook geheugen nodig dat niet gebonden is aan één specifieke functiecall. Je hebt een plaats nodig voor grotere structuren die je kunt doorgeven tussen functies. Hiervoor gebruiken veel implementaties van programmeertalen een stuk geheugen dat we de **heap** noemen. Hierin staan datastructuren met een levensduur los van die van functies. Een stuk geheugen in de heap moet daarom expliciet gereserveerd worden. Daarnaast moet het geheugen ook weer vrijgegeven worden wanneer je programma er mee klaar is. Soms moet je dit als programmeur explicet doen (bijv. in C), en soms zorgt de programmeertaal er voor dat dit automatisch gebeurt (bijv. in Python of Java). In dat geval moet die daar uiteraard wel een administratie voor bijhouden van wat in gebruik is en wat niet.

### 3.15.1 Memory Management

De meest eenvoudige implementatie van een heap is het gebruik van expliciete allocatie en geen ondersteuning voor het vrijgeven van geheugen tijdens het uitvoeren van een programma. Pas als het programma stopt is het gebruikte geheugen weer beschikbaar.

De heap bestaat dan uit slechts twee onderdelen:

- Een aaneengesloten gebied geheugen (te zien als grote array).
- Een adres (pointer) naar het begin van het nog "vrije" deel. Dit is initieel het begin van de heap (index 0 in de array).

Zodra een functie een stuk heap alloceert wordt de pointer opgeschoven tot direct na het zojuist gereserveerde segment.

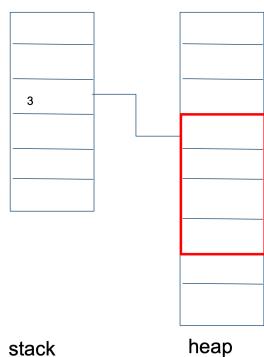
### 3.15.2 IJVM Instructies Memory Management

De IJVM machine gebruikt heeft een eenvoudige heap. Wat je op de stack kunt reserveren zijn altijd arrays van integers. Met slechts drie instructies kun je hier gebruik van maken:

---

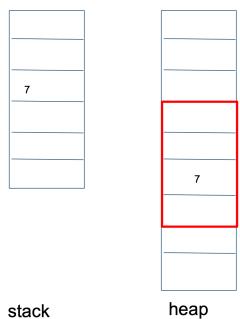
<code>new size</code>	Verwacht op de stack de grootte van het te reserveren array. Na afloop staat het adres in de heap op de stack.
<code>iaload index addr</code>	Verwacht op de stack een index in, en het adres van een array. Na afloop staat de waarde van het element in de array op de stack.
<code>iastore value index addr</code>	Verwacht op de stack een waarde, een index in, en een adres van een array. Ze na afloop niks op de stack.

---



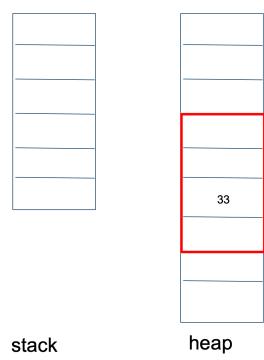
`bipush 4`  
`new`

De instructie `new` creert een blok van lengte  $n$  op de heap en zet het adres van het eerste element op de stack. Afhankelijk van de implementatie wordt ook  $n$  nog ergens opgeslagen.



`bipush 2`  
`iload 0`  
`iaload`

in `iaload` wordt de index opgeteld bij het adres en dat element uit de heap wordt op de stack gezet.



`bipush 33`  
`bipush 2`  
`iload 0`  
`iastore`

in `iastore` gebeurt het omgekeerde van `iaload`. Een waarde op de stack wordt naar de heap gekopieerd.

Enkele voorbeelden van Python constructies en de bijbehorende vertaling naar IJVM instructies zijn bijvoorbeeld:

---

```
a = [0] * 10
      bipush 10
      new
      istore 0
```

---

```
a[3] = 12
      bipush 12
      bipush 3
      iload 0
      iastore
```

---

```
b = a[3]
      bipush 3
      iload 0
      iaload
      istore 1
```

---

Let op: aan een waarde op de stack is niet te zien of dit gewoon een getal of een index naar een array (pointer) op de heap is. Beide zijn gewoon getallen. Bekijk bijvoorbeeld:

```
push 3
push 3
iaload
```

Dit geeft gewoon het 7e (tel vanaf 0) element van de heap!

In Python kan dit niet gebeuren, want alle array operaties worden gecontroleerd (door extra instructies te genereren).

Moderne programmeertalen beschermen de toegang tot de heap:

- iedere heap-verwijzing op de stack is gemarkerd en dus herkenbaar (extra bit oid)
- ieder heap-blok bevat administratie over grootte e.d.
- ieder access wordt gecontroleerd

In C(++) gebeurt dat niet en ben je dus in staat op willekeurige plekken in het geheugen een waarde te lezen of te schrijven.

### 3.15.3 Volledig voorbeeld: Selection Sort

Ter illustratie geven we een vertaling van het selection sort algoritme in Python naar IJVM instructies.

```

def test():
    a = [0] * 5
    n = 5
    a[0] = 6
    a[1] = 9
    a[2] = 3
    a[3] = 7
    a[4] = 2

    print(selsort(a,5))

def selsort(a,n):
    i = 0
    while i < n:
        m = i
        j = i + 1
        while j < n:
            if a[j]< a[m]:
                m = j
            j = j + 1
        h = a[i]
        a[i] = a[m]
        a[m] = h
        i= i + 1
    return a

call 0 2 5
pop
stop
5 bipush 5 new
istore 0
bipush 5
istore 1
bipush 6
bipush 0
iload 0
iastore
bipush 9
bipush 1
iload 0
iastore
bipush 3
bipush 2
iload 0
iastore
bipush 7
bipush 3
iload 0
iastore
bipush 2
bipush 4
iload 0
iastore
          10 bipush 0
          20 iload 2
          30 iload 3
          40 iload 2
          50 iload 3
          60 iload 4
          70 iload 5
          80 iload 6
          90 iload 7
          100 iload 8
          110 iload 9
          120 iload 10
          130 iload 11
          140 iload 12
          150 iload 13
          160 iload 14
          170 iload 15
          180 iload 16
          190 iload 17
          200 iload 18
          210 iload 19
          220 iload 20
          230 iload 21
          240 iload 22
          250 iload 23
          260 iload 24
          270 iload 25
          280 iload 26
          290 iload 27
          300 iload 28
          310 iload 29
          320 iload 30
          330 iload 31
          340 iload 32
          350 iload 33
          360 iload 34
          370 iload 35
          380 iload 36
          390 iload 37
          400 iload 38
          410 iload 39
          420 iload 40
          430 iload 41
          440 iload 42
          450 iload 43
          460 iload 44
          470 iload 45
          480 iload 46
          490 iload 47
          500 iload 48
          510 iload 49
          520 iload 50
          530 iload 51
          540 iload 52
          550 iload 53
          560 iload 54
          570 iload 55
          580 iload 56
          590 iload 57
          600 iload 58
          610 iload 59
          620 iload 60
          630 iload 61
          640 iload 62
          650 iload 63
          660 iload 64
          670 iload 65
          680 iload 66
          690 iload 67
          700 iload 68
          710 iload 69
          720 iload 70
          730 iload 71
          740 iload 72
          750 iload 73
          760 iload 74
          770 iload 75
          780 iload 76
          790 iload 77
          800 iload 78
          810 iload 79
          820 iload 80
          830 iload 81
          840 iload 82
          850 iload 83
          860 iload 84
          870 iload 85
          880 iload 86
          890 iload 87
          900 iload 88
          910 iload 89
          920 iload 90
          930 iload 91
          940 iload 92
          950 iload 93
          960 iload 94
          970 iload 95
          980 iload 96
          990 iload 97
          1000 iload 98
          1010 iload 99
          1020 iload 100
          1030 iload 101
          1040 iload 102
          1050 iload 103
          1060 iload 104
          1070 iload 105
          1080 iload 106
          1090 iload 107
          1100 iload 108
          1110 iload 109
          1120 iload 110
          1130 iload 111
          1140 iload 112
          1150 iload 113
          1160 iload 114
          1170 iload 115
          1180 iload 116
          1190 iload 117
          1200 iload 118
          1210 iload 119
          1220 iload 120
          1230 iload 121
          1240 iload 122
          1250 iload 123
          1260 iload 124
          1270 iload 125
          1280 iload 126
          1290 iload 127
          1300 iload 128
          1310 iload 129
          1320 iload 130
          1330 iload 131
          1340 iload 132
          1350 iload 133
          1360 iload 134
          1370 iload 135
          1380 iload 136
          1390 iload 137
          1400 iload 138
          1410 iload 139
          1420 iload 140
          1430 iload 141
          1440 iload 142
          1450 iload 143
          1460 iload 144
          1470 iload 145
          1480 iload 146
          1490 iload 147
          1500 iload 148
          1510 iload 149
          1520 iload 150
          1530 iload 151
          1540 iload 152
          1550 iload 153
          1560 iload 154
          1570 iload 155
          1580 iload 156
          1590 iload 157
          1600 iload 158
          1610 iload 159
          1620 iload 160
          1630 iload 161
          1640 iload 162
          1650 iload 163
          1660 iload 164
          1670 iload 165
          1680 iload 166
          1690 iload 167
          1700 iload 168
          1710 iload 169
          1720 iload 170
          1730 iload 171
          1740 iload 172
          1750 iload 173
          1760 iload 174
          1770 iload 175
          1780 iload 176
          1790 iload 177
          1800 iload 178
          1810 iload 179
          1820 iload 180
          1830 iload 181
          1840 iload 182
          1850 iload 183
          1860 iload 184
          1870 iload 185
          1880 iload 186
          1890 iload 187
          1900 iload 188
          1910 iload 189
          1920 iload 190
          1930 iload 191
          1940 iload 192
          1950 iload 193
          1960 iload 194
          1970 iload 195
          1980 iload 196
          1990 iload 197
          2000 iload 198
          2010 iload 199
          2020 iload 200
          2030 iload 201
          2040 iload 202
          2050 iload 203
          2060 iload 204
          2070 iload 205
          2080 iload 206
          2090 iload 207
          2100 iload 208
          2110 iload 209
          2120 iload 210
          2130 iload 211
          2140 iload 212
          2150 iload 213
          2160 iload 214
          2170 iload 215
          2180 iload 216
          2190 iload 217
          2200 iload 218
          2210 iload 219
          2220 iload 220
          2230 iload 221
          2240 iload 222
          2250 iload 223
          2260 iload 224
          2270 iload 225
          2280 iload 226
          2290 iload 227
          2300 iload 228
          2310 iload 229
          2320 iload 230
          2330 iload 231
          2340 iload 232
          2350 iload 233
          2360 iload 234
          2370 iload 235
          2380 iload 236
          2390 iload 237
          2400 iload 238
          2410 iload 239
          2420 iload 240
          2430 iload 241
          2440 iload 242
          2450 iload 243
          2460 iload 244
          2470 iload 245
          2480 iload 246
          2490 iload 247
          2500 iload 248
          2510 iload 249
          2520 iload 250
          2530 iload 251
          2540 iload 252
          2550 iload 253
          2560 iload 254
          2570 iload 255
          2580 iload 256
          2590 iload 257
          2600 iload 258
          2610 iload 259
          2620 iload 260
          2630 iload 261
          2640 iload 262
          2650 iload 263
          2660 iload 264
          2670 iload 265
          2680 iload 266
          2690 iload 267
          2700 iload 268
          2710 iload 269
          2720 iload 270
          2730 iload 271
          2740 iload 272
          2750 iload 273
          2760 iload 274
          2770 iload 275
          2780 iload 276
          2790 iload 277
          2800 iload 278
          2810 iload 279
          2820 iload 280
          2830 iload 281
          2840 iload 282
          2850 iload 283
          2860 iload 284
          2870 iload 285
          2880 iload 286
          2890 iload 287
          2900 iload 288
          2910 iload 289
          2920 iload 290
          2930 iload 291
          2940 iload 292
          2950 iload 293
          2960 iload 294
          2970 iload 295
          2980 iload 296
          2990 iload 297
          3000 iload 298
          3010 iload 299
          3020 iload 300
          3030 iload 301
          3040 iload 302
          3050 iload 303
          3060 iload 304
          3070 iload 305
          3080 iload 306
          3090 iload 307
          3100 iload 308
          3110 iload 309
          3120 iload 310
          3130 iload 311
          3140 iload 312
          3150 iload 313
          3160 iload 314
          3170 iload 315
          3180 iload 316
          3190 iload 317
          3200 iload 318
          3210 iload 319
          3220 iload 320
          3230 iload 321
          3240 iload 322
          3250 iload 323
          3260 iload 324
          3270 iload 325
          3280 iload 326
          3290 iload 327
          3300 iload 328
          3310 iload 329
          3320 iload 330
          3330 iload 331
          3340 iload 332
          3350 iload 333
          3360 iload 334
          3370 iload 335
          3380 iload 336
          3390 iload 337
          3400 iload 338
          3410 iload 339
          3420 iload 340
          3430 iload 341
          3440 iload 342
          3450 iload 343
          3460 iload 344
          3470 iload 345
          3480 iload 346
          3490 iload 347
          3500 iload 348
          3510 iload 349
          3520 iload 350
          3530 iload 351
          3540 iload 352
          3550 iload 353
          3560 iload 354
          3570 iload 355
          3580 iload 356
          3590 iload 357
          3600 iload 358
          3610 iload 359
          3620 iload 360
          3630 iload 361
          3640 iload 362
          3650 iload 363
          3660 iload 364
          3670 iload 365
          3680 iload 366
          3690 iload 367
          3700 iload 368
          3710 iload 369
          3720 iload 370
          3730 iload 371
          3740 iload 372
          3750 iload 373
          3760 iload 374
          3770 iload 375
          3780 iload 376
          3790 iload 377
          3800 iload 378
          3810 iload 379
          3820 iload 380
          3830 iload 381
          3840 iload 382
          3850 iload 383
          3860 iload 384
          3870 iload 385
          3880 iload 386
          3890 iload 387
          3900 iload 388
          3910 iload 389
          3920 iload 390
          3930 iload 391
          3940 iload 392
          3950 iload 393
          3960 iload 394
          3970 iload 395
          3980 iload 396
          3990 iload 397
          4000 iload 398
          4010 iload 399
          4020 iload 400
          4030 iload 401
          4040 iload 402
          4050 iload 403
          4060 iload 404
          4070 iload 405
          4080 iload 406
          4090 iload 407
          4100 iload 408
          4110 iload 409
          4120 iload 410
          4130 iload 411
          4140 iload 412
          4150 iload 413
          4160 iload 414
          4170 iload 415
          4180 iload 416
          4190 iload 417
          4200 iload 418
          4210 iload 419
          4220 iload 420
          4230 iload 421
          4240 iload 422
          4250 iload 423
          4260 iload 424
          4270 iload 425
          4280 iload 426
          4290 iload 427
          4300 iload 428
          4310 iload 429
          4320 iload 430
          4330 iload 431
          4340 iload 432
          4350 iload 433
          4360 iload 434
          4370 iload 435
          4380 iload 436
          4390 iload 437
          4400 iload 438
          4410 iload 439
          4420 iload 440
          4430 iload 441
          4440 iload 442
          4450 iload 443
          4460 iload 444
          4470 iload 445
          4480 iload 446
          4490 iload 447
          4500 iload 448
          4510 iload 449
          4520 iload 450
          4530 iload 451
          4540 iload 452
          4550 iload 453
          4560 iload 454
          4570 iload 455
          4580 iload 456
          4590 iload 457
          4600 iload 458
          4610 iload 459
          4620 iload 460
          4630 iload 461
          4640 iload 462
          4650 iload 463
          4660 iload 464
          4670 iload 465
          4680 iload 466
          4690 iload 467
          4700 iload 468
          4710 iload 469
          4720 iload 470
          4730 iload 471
          4740 iload 472
          4750 iload 473
          4760 iload 474
          4770 iload 475
          4780 iload 476
          4790 iload 477
          4800 iload 478
          4810 iload 479
          4820 iload 480
          4830 iload 481
          4840 iload 482
          4850 iload 483
          4860 iload 484
          4870 iload 485
          4880 iload 486
          4890 iload 487
          4900 iload 488
          4910 iload 489
          4920 iload 490
          4930 iload 491
          4940 iload 492
          4950 iload 493
          4960 iload 494
          4970 iload 495
          4980 iload 496
          4990 iload 497
          5000 iload 498
          5010 iload 499
          5020 iload 500
          5030 iload 501
          5040 iload 502
          5050 iload 503
          5060 iload 504
          5070 iload 505
          5080 iload 506
          5090 iload 507
          5100 iload 508
          5110 iload 509
          5120 iload 510
          5130 iload 511
          5140 iload 512
          5150 iload 513
          5160 iload 514
          5170 iload 515
          5180 iload 516
          5190 iload 517
          5200 iload 518
          5210 iload 519
          5220 iload 520
          5230 iload 521
          5240 iload 522
          5250 iload 523
          5260 iload 524
          5270 iload 525
          5280 iload 526
          5290 iload 527
          5300 iload 528
          5310 iload 529
          5320 iload 530
          5330 iload 531
          5340 iload 532
          5350 iload 533
          5360 iload 534
          5370 iload 535
          5380 iload 536
          5390 iload 537
          5400 iload 538
          5410 iload 539
          5420 iload 540
          5430 iload 541
          5440 iload 542
          5450 iload 543
          5460 iload 544
          5470 iload 545
          5480 iload 546
          5490 iload 547
          5500 iload 548
          5510 iload 549
          5520 iload 550
          5530 iload 551
          5540 iload 552
          5550 iload 553
          5560 iload 554
          5570 iload 555
          5580 iload 556
          5590 iload 557
          5600 iload 558
          5610 iload 559
          5620 iload 560
          5630 iload 561
          5640 iload 562
          5650 iload 563
          5660 iload 564
          5670 iload 565
          5680 iload 566
          5690 iload 567
          5700 iload 568
          5710 iload 569
          5720 iload 570
          5730 iload 571
          5740 iload 572
          5750 iload 573
          5760 iload 574
          5770 iload 575
          5780 iload 576
          5790 iload 577
          5800 iload 578
          5810 iload 579
          5820 iload 580
          5830 iload 581
          5840 iload 582
          5850 iload 583
          5860 iload 584
          5870 iload 585
          5880 iload 586
          5890 iload 587
          5900 iload 588
          5910 iload 589
          5920 iload 590
          5930 iload 591
          5940 iload 592
          5950 iload 593
          5960 iload 594
          5970 iload 595
          5980 iload 596
          5990 iload 597
          6000 iload 598
          6010 iload 599
          6020 iload 600
          6030 iload 601
          6040 iload 602
          6050 iload 603
          6060 iload 604
          6070 iload 605
          6080 iload 606
          6090 iload 607
          6100 iload 608
          6110 iload 609
          6120 iload 610
          6130 iload 611
          6140 iload 612
          6150 iload 613
          6160 iload 614
          6170 iload 615
          6180 iload 616
          6190 iload 617
          6200 iload 618
          6210 iload 619
          6220 iload 620
          6230 iload 621
          6240 iload 622
          6250 iload 623
          6260 iload 624
          6270 iload 625
          6280 iload 626
          6290 iload 627
          6300 iload 628
          6310 iload 629
          6320 iload 630
          6330 iload 631
          6340 iload 632
          6350 iload 633
          6360 iload 634
          6370 iload 635
          6380 iload 636
          6390 iload 637
          6400 iload 638
          6410 iload 639
          6420 iload 640
          6430 iload 641
          6440 iload 642
          6450 iload 643
          6460 iload 644
          6470 iload 645
          6480 iload 646
          6490 iload 647
          6500 iload 648
          6510 iload 649
          6520 iload 650
          6530 iload 651
          6540 iload 652
          6550 iload 653
          6560 iload 654
          6570 iload 655
          6580 iload 656
          6590 iload 657
          6600 iload 658
          6610 iload 659
          6620 iload 660
          6630 iload 661
          6640 iload 662
          6650 iload 663
          6660 iload 664
          6670 iload 665
          6680 iload 666
          6690 iload 667
          6700 iload 668
          6710 iload 669
          6720 iload 670
          6730 iload 671
          6740 iload 672
          6750 iload 673
          6760 iload 674
          6770 iload 675
          6780 iload 676
          6790 iload 677
          6800 iload 678
          6810 iload 679
          6820 iload 680
          6830 iload 681
          6840 iload 682
          6850 iload 683
          6860 iload 684
          6870 iload 685
          6880 iload 686
          6890 iload 687
          6900 iload 688
          6910 iload 689
          6920 iload 690
          6930 iload 691
          6940 iload 692
          6950 iload 693
          6960 iload 694
          6970 iload 695
          6980 iload 696
          6990 iload 697
          7000 iload 698
          7010 iload 699
          7020 iload 700
          7030 iload 701
          7040 iload 702
          7050 iload 703
          7060 iload 704
          7070 iload 705
          7080 iload 706
          7090 iload 707
          7100 iload 708
          7110 iload 709
          7120 iload 710
          7130 iload 711
          7140 iload 712
          7150 iload 713
          7160 iload 714
          7170 iload 715
          7180 iload 716
          7190 iload 717
          7200 iload 718
          7210 iload 719
          7220 iload 720
          7230 iload 721
          7240 iload 722
          7250 iload 723
          7260 iload 724
          7270 iload 725
          7280 iload 726
          7290 iload 727
          7300 iload 728
          7310 iload 729
          7320 iload 730
          7330 iload 731
          7340 iload 732
          7350 iload 733
          7360 iload 734
          7370 iload 735
          7380 iload 736
          7390 iload 737
          7400 iload 738
          7410 iload 739
          7420 iload 740
          7430 iload 741
          7440 iload 742
          7450 iload 743
          7460 iload 744
          7470 iload 745
          7480 iload 746
          7490 iload 747
          7500 iload 748
          7510 iload 749
          7520 iload 750
          7530 iload 751
          7540 iload 752
          7550 iload 753
          7560 iload 754
          7570 iload 755
          7580 iload 756
          7590 iload 757
          7600 iload 758
          7610 iload 759
          7620 iload 760
          7630 iload 761
          7640 iload 762
          7650 iload 763
          7660 iload 764
          7670 iload 765
          7680 iload 766
          7690 iload 767
          7700 iload 768
          7710 iload 769
          7720 iload 770
          7730 iload 771
          7740 iload 772
          7750 iload 773
          7760 iload 774
          7770 iload 775
          7780 iload 776
          7790 iload 777
          7800 iload 778
          7810 iload 779
          7820 iload 780
          7830 iload 781
          7840 iload 782
          7850 iload 783
          7860 iload 784
          7870 iload 785
          7880 iload 786
          7890 iload 787
          7900 iload 788
          7910 iload 789
          7920 iload 790
          7930 iload 791
          7940 iload 792
          7950 iload 793
          7960 iload 794
          7970 iload 795
          7980 iload 796
          7990 iload 797
          8000 iload 798
          8010 iload 799
          8020 iload 800
          8030 iload 801
          8040 iload 802
          8050 iload 803
          8060 iload 804
          8070 iload 805
          8080 iload 806
          8090 iload 807
          8100 iload 808
          8110 iload 809
          8120 iload 810
          8130 iload 811
          8140 iload 812
          8150 iload 813
          8160 iload 814
          8170 iload 815
          8180 iload 816
          8190 iload 817
          8200 iload 818
          8210 iload 819
          8220 iload 820
          8230 iload 821
          8240 iload 822
          8250 iload 823
          8260 iload 824
          8270 iload 825
          8280 iload 826
          8290 iload 827
          8300 iload 828
          8310 iload 829
          8320 iload 830
          8330 iload 831
          8340 iload 832
          8350 iload 833
          8360 iload 834
          8370 iload 835
          8380 iload 836
          8390 iload 837
          8400 iload 838
          8410 iload 839
          8420 iload 840
          8430 iload 841
          8440 iload 842
          8450 iload 843
          8460 iload
```

Er zijn meerdere strategieen voor garbage collection. Python maakt gebruik van “reference counting”. Daarbij wordt voor ieder object op de heap bijgehouden hoeveel variabelen, datastructuren en functies er een verwijzing naar hebben. Deze teller wordt automatisch opgehoogd en verlaagd bij het toekennen van variabelen, of het aanroepen of returnen uit een functie. Zodra de teller op nul komt te staan mag de ruimte van het object hergebruikt worden.

Een taal als Java gebruikt normaalgesproken geen reference counting. In plaats daarvan wordt het programma gepauzeerd zodra de hoeveel vrij geheugen beneden een kritieke grens komt. Vervolgens wordt de gehele stack afgelopen om te markeren welke objecten direct of indirect benaderbaar zijn vanaf de variabelen op de stack. Alle overgebleven objecten zijn niet meer bereikbaar en kunnen dus kan hun ruimte hergebruikt worden. Een belangrijke voorwaarde hiervoor is wel dat je op de stack het verschil moet kunnen zien tussen adressen op de heap en gewone integers. In de IJVM is dit dus niet zomaar mogelijk.

### 3.16 Recursieve functies

Recursieve functies zijn functies die zichzelf aanroepen. Dit blijkt een krachtige manier te zijn om bepaalde programmeerproblemen op te lossen. In principe heb je zelfs geen (while of for) loops nodig als je recursie kunt toepassen. Je kunt een loop namelijk altijd ook als een recursie schrijven.

Bekijk bijvoorbeeld een functie die de faculteit uitrekent:

```
def facloop(n):
    res = 1
    k = 1
    while k <= n:
        res = res * k
        k = k + 1
    return res
```

Dezelfde functie kun je ook recursief uitrekenen:

```
def facrec(n):
    if n == 0:
        return 1
    else:
        return n * facrec(n - 1)
```

Recursieve functies gebruiken impliciet de stack om in een groot probleem bij te houden waar ze gebonden zijn terwijl ze steeds hetzelfde recept toe passen op steeds kleinere deelproblemen. Daardoor worden sommige problemen eenvoudiger (en eleganter) om te beschrijven. De recursieve definitie van de faculteitfunctie gebruikt geen extra lokale variabelen en ligt dicht bij de wiskundige definitie van wat de faculteit is.

Het is niet altijd zo eenvoudig, maar het kan altijd. Bekijk bijvoorbeeld de functie om het kleinste element in een lijst te zoeken:

```
def smallest_loop(a):
    s = a[0]
    n = len(a)
    k = 1
    while k < n:
        if a[k] < s:
            s = a[k]
        k + 1
    return s
```

Dit kun je ook recursief aanpakken:

```
def smallest_rec(a,s,k):
    if k == len(a):
        return s
    elif a[k] < s:
        return smallest_rec(a,a[k],k+1)
    else:
        return smallest_rec(a,s,k+1)
```

In plaats van met een loop de hele array af te lopen, wordt hier aan de functie meegegeven welk deel van de lijst nog doorzocht moet worden en wat tot nu toe het kleinste gevonden element is. Hier is de loop-versie eenvoudiger, maar laat de recursieve oplossing zien hoe je door het probleem steeds iets kleiner te maken bij elke aanroep ook alle elementen van een lijst kunt bekijken.

In het algemeen is het omzetten van een loop naar een recursie niet moeilijk. Je past hetzelfde patroon toe als in `smallest_rec`. Het omzetten van een recursieve oplossing naar een iteratieve oplossing is niet altijd zo makkelijk.

### 3.16.1 Voorbeeld: de Torens van Hanoi

Als voorbeeld van een moeilijk probleem dat eenvoudig recursief te beschrijven is bekijken we de Torens van Hanoi.

In dit probleem moet je een piramide van  $n$  schijven van oplopende grootte verplaatsen. Er zijn een paar spelregels:

- Je mag maar 1 schijf per keer verplaatsen
- Er mag nooit een grotere schijf op een kleinere geplaatst worden

De schijven beginnen op volgorde op de beginstapel en moeten op volgorde eindigen op de eindstapel. Je mag tussendoor gebruik maken van slechts 1 hulpstapel.

Dit probleem lijkt eenvoudig, maar het aantal verplaatsingen dat je moet doen loopt snel op bij grotere  $n$ . Het is namelijk  $2^n - 1$ !



We willen nu een programma maken dat alle verplaatsingen print. Bijvoorbeeld 1 → 3, 1 → 2, 3 → 2 etc.

Dit kan iteratief opgelost worden, maar dit is heel lastig! De recursieve oplossing is veel eenvoudiger.

De aanpak werkt net als bij een inductiebewijs in de logica. Gegeven een probleem van grootte  $n$  moet je twee dingen oplossen:

- Wat moet je doen als  $n = 1$  (of 0, dat hangt van het probleem af)?

- Hoe reduceer je het probleem van grootte  $n$  tot een probleem van grootte  $n - 1$ ?

Bij de Torens van Hanoi is het geval van  $n = 0$  (aantal schijven) triviaal. Als we geen schijven hebben hoeven we niets te doen!

We hoeven dus alleen te bedenken wat we moeten doen als we aannemen dat we het probleem al opgelost hebben voor  $n - 1$  schijven.

Als we de 3 mogelijke locaties a, b en c noemen, dan mogen we dus aannemen dat we een willekeurige stapel van hoogte  $n - 1$  van a naar b kunnen verplaatsen met locatie c als hulpstapel.

De stappen zijn dan als volgt:

1. Verplaats eerst de bovenste  $n - 1$  schijven van locatie a naar locatie c met locatie b als hulp.
2. Verplaats nu de bovenste schijf van locatie a naar b.
3. Verplaats nu de  $n - 1$  schijven van locatie c naar paal b met locatie a als hulp.

In een Python functie is dit heel eenvoudig te beschrijven:

```
def hanoi(n,a,b,c):
    if n > 0:
        hanoi(n-1,a,c,b)
        print(a, '->', b)
        hanoi(n-1,c,b,a)

hanoi(5,1,2,3)
```

Als we dit uitschrijven voor 3 schijven krijgen we:

```
hanoi(3,1,2,3)
hanoi(2,1,3,2) + (1,2) + hanoi(2,3,2,1)
hanoi(1,1,2,3) + (1,3) + hanoi(1,2,3,1) + (1,2) + hanoi(2,3,2,1)
(1,2) + (1,3) + (2,3) + (1,2) + hanoi(2,3,2,1)
(1,2) + (1,3) + (2,3) + (1,2) + hanoi(1,3,1,2) + (3,2) + hanoi(1,1,2,3)
(1,2) + (1,3) + (2,3) + (1,2) + (3,1) + (3,2) + (1,2)
```

### 3.16.2 Voorbeeld Subsets

Een ander probleem wat mooi recursief op te lossen is, is het bepalen van alle deelverzamelingen van een verzameling.

Bekijk bijvoorbeeld alle deelverzamelingen van [1,2,3]:

```
as = [ [], [1], [2], [3], [1,2], [1,3], [2,3], [1,2,3] ]
```

Een “kleinere” versie van dit probleem zijn de deelverzamelingen van [2,3]:

```
bs = [ [], [2], [3], [2,3] ]
```

Als we het verschil tussen bs en as bekijken zijn dat de elementen:

```
diff_as_bs = [ [1], [1,2], [1,3], [1,2,3] ]
```

Dit zijn alle elementen van bs met daar een 1 aan toegevoegd. We kunnen as dus construeren uit bs als:  $bs + [[1] + s \text{ for } s \text{ in } bs]$ .

Daarmee hebben we dus het recept voor  $n$  gegeven  $n - 1$ . Wat we nu nog nodig hebben is het basisgeval voor  $n = 0$ . In dit geval is dat de lege verzameling. De verzameling deelverzamelingen van de lege verzameling is de singleton verzameling met alleen de lege verzameling.

```
subsets([]) == []
```

Wanneer we deze beide gevallen combineren krijgen we de volgende python functie:

```
def subsets(k, xs):
    if k == len(xs):
        return []
    else:
        h = a[k]
        ss = subsets(k+1, xs)
        return ss + [[h] + s for s in ss]

print(subsets(0, [1, 2, 3, 4]))
```

### 3.16.3 Algemene aanpak

Bij het schrijven van een recursieve functie volg je dus eigenlijk hetzelfde patroon als bij een inductiebewijs:

- De functie moet een basisgeval hebben waarbij het antwoord triviaal is.
- Bij iedere recursieve aanroep moet je het probleem kleiner maken (dichter bij het basisgeval)
- Je moet vertrouwen dat de recursieve aanroep je een correct antwoord geeft

Daarnaast moet je opletten dat je geen “dubbel” werk doet. Als je een recursieve functie maakt die voor  $n$  iets uitrekent dat je voor  $n - 1$  ook al had uitgerekend krijg je heel snel heel veel zinloze rekentijd.

## 3.17 Recursieve structuren

Recurse kom je niet alleen tegen in functies, maar ook in (data)structuren. Een goed voorbeeld daarvan is de mappenstructuur van het bestandsysteem van je computer. Iedere map kan zowel bestanden als ook weer andere mappen bevatten. Die mappen kunnen ook weer mappen bevatten die zelf ook weer mappen bevatten enzovoorts.

Wanneer je een programma moet schrijven waarin een dergelijke structuur voorkomt ligt het ook weer voor de hand om recursieve functies te gebruiken die de structuur volgen.

Wanneer we bijvoorbeeld het aantal bestanden op disk willen tellen is het basisgeval een lege map, of een map waarin enkel bestanden staan. Voor mappen die zelf mappen bevatten doe je voor iedere map een recursieve aanroep om het aantal bestanden in die map te bepalen.

# 4. Opdrachten en Huiswerk

In dit hoofdstuk vind je alle opdrachten. Wanneer voor een opdracht extra materiaal nodig is kun je dat normaalgesproken op de ELO vinden.

## 4.1 Les 1: Eindige automaten en reguliere expressies 1

### Tijdens de les

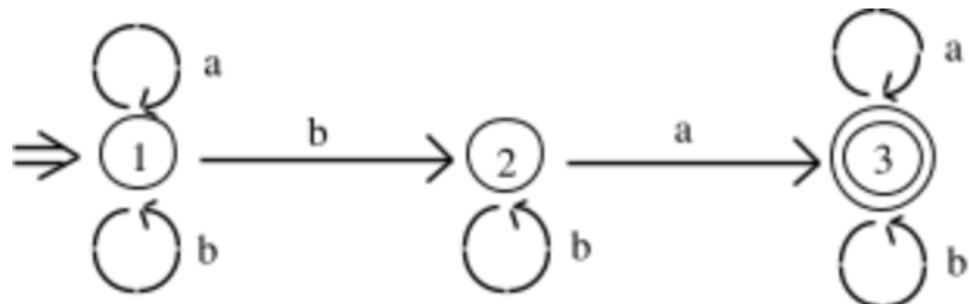
#### 1.1 Eindige automaat maken (Oefenen)

Maak een eindige automaat voor de volgende talen van strings bestaande uit a's en b's:

- (a) alle strings waarin precies 2 a's voorkomen
- (b) alle strings waarin na een a altijd een of meer b's komen
- (c) alle strings die bestaan uit een even aantal a's gevolgd door een even aantal b's
- (d) alle strings waarin ab niet voorkomt
- (e) alle strings waarin aaa precies 2 keer voorkomt (in aaaa is dit ook het geval)

#### 1.2 Eindige automaat analyseren (Oefenen)

Gegeven is onderstaande eindige automaat:



- (a) Is de automaat deterministisch
- (b) Beschrijf de taal van de automaat

### Na de les

#### 1.3 Eindige automaten maken (Oefenen)

Maak een eindige automaat voor de volgende talen van strings bestaande uit a's en b's door de betekenis van de toestanden goed te definiëren:

- (a) alle strings waarin precies 3 b's voorkomen
- (b) alle strings waarin na een a altijd precies twee b's komen
- (c) alle strings waarin een even aantal a's en een even aantal b's voorkomt

Je kunt hierbij gebruik maken van de online tool op:

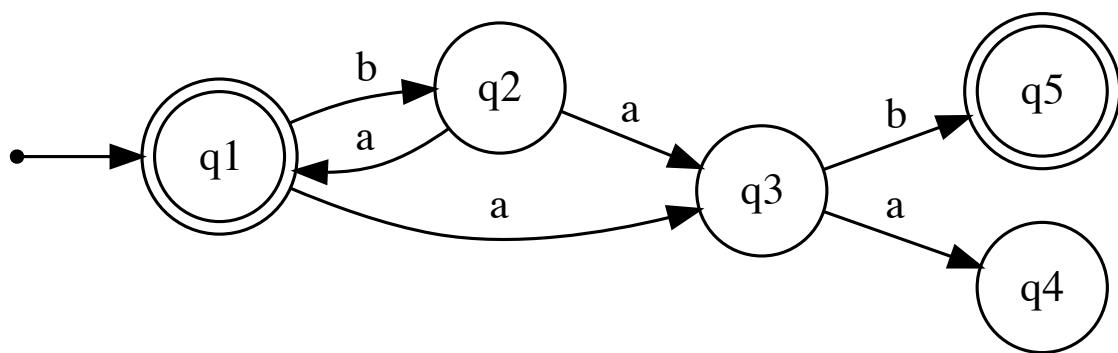
<https://tvit24.cloud.c2lab.nl/fsm.html>

## 4.2 Les 2: Eindige automaten en reguliere expressies 2

### Tijdens de les

#### 2.1 Oefenen deterministisch maken (Oefenen)

Gegeven is onderstaande automaat:



Maak een deterministische versie van deze automaat.

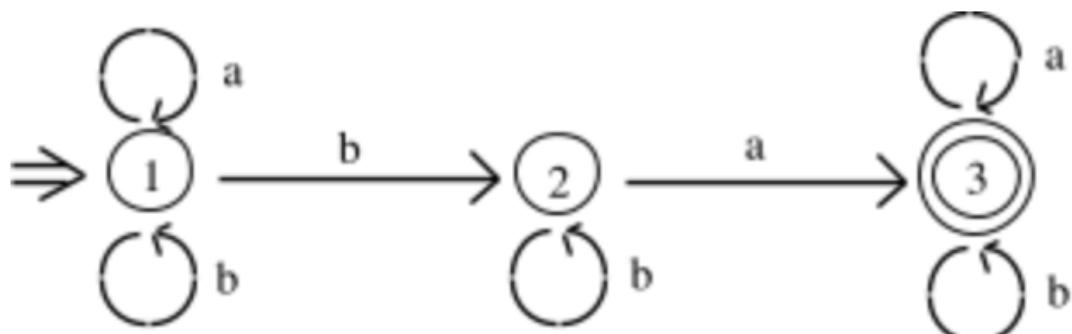
#### 2.2 Oefenen reguliere expressie omzetten (Oefenen)

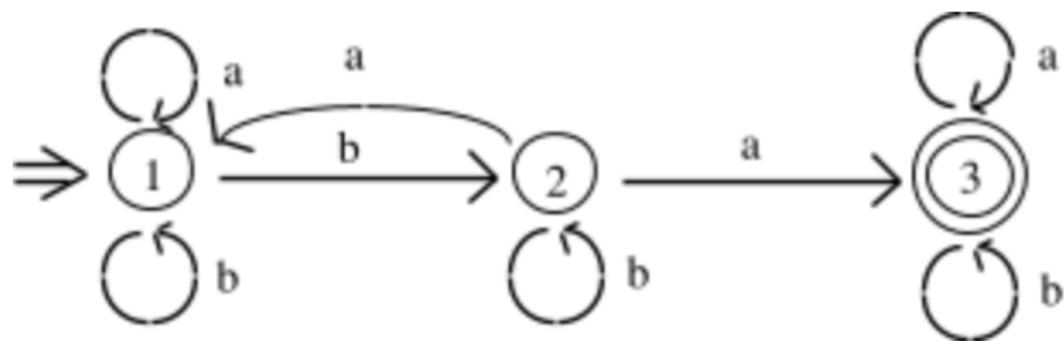
Je krijgt nadere instructie van de docent.

### Na de les

#### 2.3 Deterministisch maken (Oefenen)

Maak equivalente deterministische eindige automaten voor de volgende automaten:





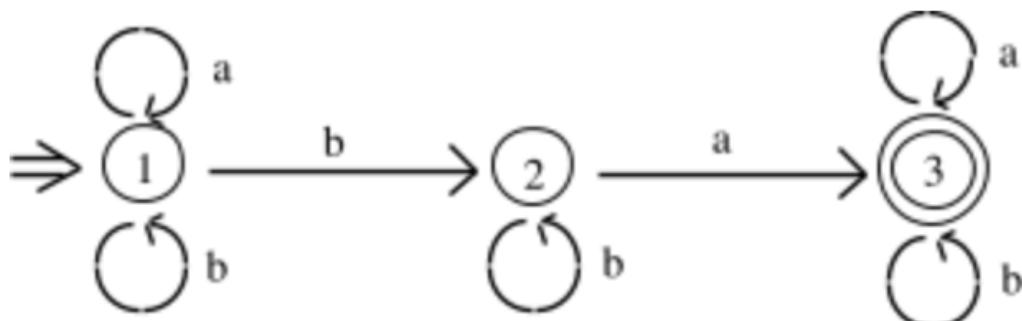
## 2.4 Deterministisch maken (Oefenen)

- (a) Maak een nondeterministische eindige automaat voor de taal van alle strings bestaande uit a's en b's waarin aabaab voorkomt
- (b) Maak voor de automaat uit onderdeel a. een equivalente deterministische automaat

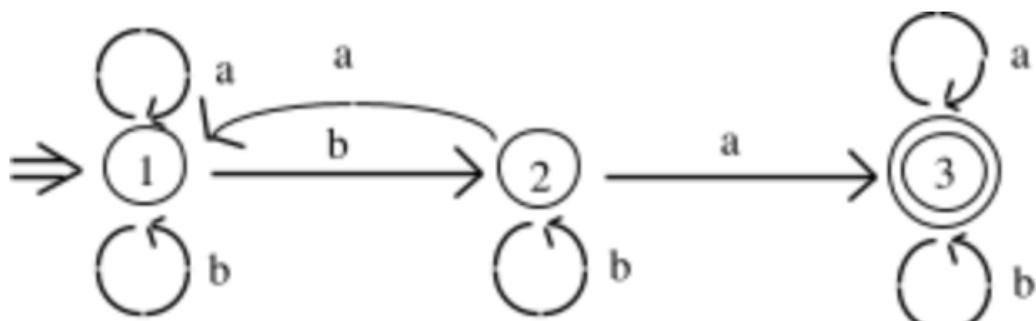
## 2.5 Van automaat naar expressie (Oefenen)

Bepaal reguliere expressies voor de talen van de volgende automaten:

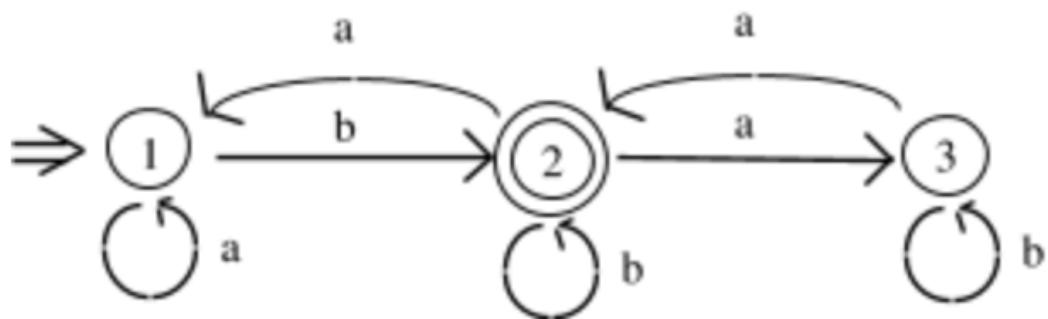
(a)



(b)



(c)



### 2.6 Van expressie naar automaat (Oefenen)

Bepaal eindige automaten bij de volgende reguliere expressies:

- (a)  $(a \mid b)^* ab(aa)^*$
- (b)  $(a \mid b)^* (aa)^*$
- (c)  $(a (aab)^*)^* \mid (bb)^*$

## 4.3 Les 3: Pushdown automaten en grammatica's 1

### Tijdens de les

#### 3.1 Gelijk delen (Oefenen)

Maak een eindige automaat voor een aantal keer het symbool a, gevolgd door even vaak het symbool b.

#### 3.2 Oefenen pushdown automaten (Oefenen)

Maak een pushdown-automaat voor:

- (a)  $\{a^i b^j c^k \mid i + j = k\}$
- (b)  $\{a^i b^j c^k \mid i + k = j\}$

### Na de les

#### 3.3 Pushdown-automaten maken (Oefenen)

Maak een pushdown-automaat voor:

- (a)  $\{a^i b^j c^k \mid i = j \text{ of } j = k\}$
- (b)  $\{a^i b^j \mid i \neq j\}$

#### 3.4 Even veel (Oefenen)

Gegeven is de volgende grammatica:  $X = ab \mid a X b$

Geef ontleedbomen voor de volgende strings:

- (a) aabb
- (b) aaabbb
- (c) aaaaaaabbbbbbb
- (d) beredeneer dat de taal van  $X = \{a^n b^n \mid n \geq 1\}$  waarbij  $a^n$  betekent n a's achter elkaar

## 4.4 Les 4: Pushdown automaten en grammatica's 2

### Tijdens de les

#### 4.1 Oefenen met grammatica's (Oefenen)

Je oefent zelf verder met het huiswerk.

### Na de les

#### 4.2 Abc's (Oefenen)

Gegeven is de volgende grammatica:

$$\begin{aligned} S &= X \mid c \ S \\ X &= ab \mid a \ X \ b \end{aligned}$$

Geef ontleedbomen voor de volgende strings:

- (a) caabb
- (b) ccaaabbb
- (c) cccccccab
- (d) wat is de taal van X ?

#### 4.3 Abcd's (Oefenen)

Gegeven is de volgende grammatica:

$$\begin{aligned} S &= d \ X \mid c \ S \\ X &= a \ X \mid Y \\ Y &= b \ Y \mid \end{aligned}$$

Geef ontleedbomen voor de volgende strings:

- (a) cdaa
- (b) ccdaaabbb
- (c) cccccccdab
- (d) Geef een reguliere expressie en een eindige automaat voor de taal S

#### 4.4 Aantallen matchen (Oefenen)

Stel voor de volgende talen grammatica's op ( $a^n$  betekent n a's achter elkaar):

- (a)  $\{a^n b^n \mid n \geq 0\}$
- (b)  $\{a^n b^m \mid n \geq 0 \wedge m \geq 0 \wedge m \leq n\}$
- (c)  $\{(a^n b^n)^* c^* \mid n \geq 0\}$

#### 4.5 Rekenkundige expressies ontleden (Oefenen)

De volgende grammatica kan ook alle rekenkundige expressies aan:

```
cijfer    = 0 | 1 | 2 .. | 9
getal    = cijfer | cijfer getal
operator = + | - | * | /
expressie = getal | ( expressie ) | expressie operator expressie
```

Geef zoveel mogelijk ontleedbomen voor de volgende rekenkundige expressies en vermeld ook telkens de waarde van de onderdelen:

- (a)  $89 + 007 * 3$

- (b)  $(8 - 90) / 7 * 8$   
(c)  $(78 + 9 + 78) - (6)$

Wat zou het nadeel zijn van deze grammatica?

#### 4.6 Pascal ontleden (Oefenen)

Maak de volgende (deel)grammatica voor Pascal statements af:

```

statement      = assignment
                | ifstatement
                | whilestatement
                | forstatement
                | repeatstatement
                | group
                |
group          = "begin" statements "end"
statements     = | statement | statement (";" statement)*
assignment     = variable ":=" expression
ifstatement    = "if" condition "then" statement | ...
whilestatement = ...
forstatement   = ...
repeatstatement = ...

```

Hierbij hoeven variable, expression en condition niet verder uitgewerkt te worden. Deze grammatica werkt niet op letters zoals we gewend zijn, maar op strings zoals "if" en "while" en "repeat".

## 4.5 Les 5: Turingmachines 1

### Tijdens de les

#### 5.1 Herhaling Ontleden (Oefenen)

Gegeven is de volgende grammatica:

```

program = statement*
statement = assignment | conditional
assignment = identifier ':=' value ';'
conditional = 'if (' expression ') {' statement* '}'
expression = value | expression '=' expression
value = integer | identifier
integer = number number*
number = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
identifier = letter letter*
letter = x | y | z

```

Geef een ontleedboom voor het volgende programma dat voldoet aan de regel program.

```

x := 12;
if (x = y) {
    z := 23;
}
z := 34;

```

#### 5.2 Oefening Turingmachine Palindroom (Oefenen)

Ontwerp een Turing Machine die palindromen bestaande uit 0 en 1 afgesloten met een b herkent.

## Na de les

### 5.3 Turing machines maken (eenvoudig) (Oefenen)

Algemeen: de invoer bestaat uit 0, 1 en \_ eventueel met hulpsymbolen. Je mag steeds zelf kiezen welke symbolen je schrijft. Rechts van de invoer is de tape gevuld met (oneindig veel) \_.

- (a) Check of de invoer uit een even aantal symbolen bestaat.
- (b) De invoer bestaat uit een 0-1 string afgesloten met een #. Kopieer de hele string naar de tape achter #. Zorg er ook voor dat de originele invoer weer wordt teruggezet.
- (c) Als b, maar kopieer gespiegeld.
- (d) Schuif de hele invoer 1 positie naar rechts en voeg een 0 toe aan het begin.

## 4.6 Les 6: Turingmachines 2

### Tijdens de les

#### 6.1 Oefenen (Oefenen)

Werk verder aan de huiswerkopgaven.

## Na de les

### 6.2 Turing machines maken (moeilijker) (Oefenen)

Algemeen: de invoer bestaat uit 0, 1 en \_ eventueel met hulpsymbolen. Je mag steeds zelf kiezen welke symbolen je schrijft. Rechts van de invoer is de tape gevuld met (oneindig veel) \_.

- (a) De invoer bestaat uit enen gevolgd door nullen. Accepteer als het aantal enen groter is dan het aantal nullen.
- (b) De invoer is een gespiegeld binair getal (vanrechts-naar-linksdus). Verhoog het getal met 1.
- (c) De invoer is een gespiegeld binair getal(vanrechts-naar-linksdus). Verlaag het getal met 1. Het resultaat moet dus op het eind op de tape staan.
- (d) De invoer bestaat uit een 0-1 string met daar middenin een \$ symbool. Na deze string staat een # gevolgd door nog een 0-1 string. Substitueer de tweede 0-1 string op de plaats van de \$. Tip: kopieer eerst het deel van de eerste string na het \$ naar het einde van de tape. Bijvoorbeeld: 100\$101#111 wordt 100111101

### 6.3 Bonus: Turing machines maken (nog moeilijker) (Oefenen)

Algemeen: de invoer bestaat uit 0, 1 en \_ eventueel met hulpsymbolen. Je mag steeds zelf kiezen welke symbolen je schrijft. Rechts van de invoer is de tape gevuld met (oneindig veel) \_.

- (a) De invoer bestaat uit een reeks enen (unair getal) afgesloten door een #. Na afloop staat in binaire notatie het aantal enen rechts van het # (gespiegeld).
- (b) De invoer bestaat uit 2 binaire getallen gescheiden door een # en afgesloten door een #. Na afloop staat voorbij het tweede # de som van de 2 getallen. Alle getallen mogen hierbij gespiegeld staan.
- (c) Variant op b: er is geen tweede #, het resultaat komt op de plaats van het tweede getal te staan.

## 4.7 Les 7: Uitloop 1

### Tijdens de les

#### 7.1 Herhalen automaten en expressies (Oefenen)

We kiezen een oefening op basis van het huiswerk en oefenen die samen opnieuw.

**7.2 Herhalen pushdown automaten en grammatica's (Oefenen)**

We kiezen een oefening op basis van het huiswerk en oefenen die samen opnieuw.

**7.3 Herhalen Turingmachines (Oefenen)**

We kiezen een oefening op basis van het huiswerk en oefenen die samen opnieuw.

## 4.8 Les 8: Talen afbeelden 1

### Tijdens de les

**8.1 IJVM Als rekenmachine (Oefenen)**

Bereken de volgende sommen met de IJVM simulator

- (a)  $6 + 7 + 8$
- (b)  $6 - 3 - 1$
- (c)  $6 - (3 - 1)$
- (d)  $17 * 3 * (6 + 3 - 1)$

**8.2 Oefenen functiecalls (Oefenen)**

Vertaal de volgende python functie naar IJVM instructies:

```
def dummy(x,y):
    return 42
```

Hoe ziet de stack er uit tijdens de aanroep van `dummy(4,2)`

### Na de les

**8.3 Expressies vertalen (Oefenen)**

Beschrijf de volgende expressies als reeks IJVM instructies

$6 + 7 + 8$   
 $6 - 3 - 1$   
 $6 - (3 - 1)$   
 $17 - 3 * (6 + 3 - 1)$

**8.4 Delers bepalen (Oefenen)**

Maak eerst in Python een programma `deler(k,n)` dat test of  $n$  deelbaar is door  $k$ . Doe dit door herhaaldelijk  $k$  op te tellen vanaf  $k$  tot dat je precies op  $n$  komt of er overeen gaat (geef 0 of 1 als resultaat).

Vertaal dit programma naar IJVM code.

## 4.9 Les 9: Talen afbeelden 2

### Tijdens de les

**9.1 Oefenen functies vertalen (Oefenen)**

Vertaal de volgende Python functies naar IJVM instructies

- (a) `def increment(a):`  
 `return a + 1`

```
(b) def fac(n):
    result = 1
    while n > 1:
        result = result * n
        n = n - 1
    return result
```

### 9.2 Oefenen gebruik heap (Oefenen)

Vertaal de volgende Python functies naar IJVM instructies

```
(a) def numthrees(a, len):
    num = 0
    for i in range(len):
        if a[i] == 3:
            num = num + 1
    return num

(b) def find(x, a, len):
    pos = 0
    while pos < len:
        if a[pos] == x:
            return pos
        pos = pos + 1
    return -1
```

## 4.10 Les 10: Uitloop 2

## 4.11 Les 11: Recursie en Iteratie 1

### Tijdens de les

#### 11.1 Iteratieve torens van Hanoi (Oefenen)

Bestudeer het patroon van verplaatsingen bij “de torens van Hanoi” en verzin een algoritme dat met behulp van loops de verplaatsingen opsomt.

#### 11.2 Oefenen recursie (Oefenen)

Geef voor de volgende problemen een oplossing met een while-lus en een recursieve versie.

- Maak in Python een functie `find(x, xs)` die als resultaat `True` heeft als het getal `x` in de lijst `xs` voorkomt en anders `False`.
- Maak in Python een functie `index(x, xs)` die als resultaat de eerste positie van `x` in `xs` teruggeeft. Als `x` niet voorkomt moet het resultaat `-1` zijn.
- Maak in Python een functie `isSorted(xs)` die als resultaat `True` heeft als de lijst `xs` gesorteerd is van klein naar groot en anders `False`.
- Maak in Python een functie `count(x, xs)` die als resultaat het aantal keren dat `x` voorkomt in `xs` geeft.
- Maak in Python een functie `group3(xs)` die als resultaat de lijst `xs` in groepjes van 3 oplevert (een lijst van lijsten dus). Ga er van uit dat het aantal elementen van `xs` een 3-voud is.

### Na de les

#### 11.3 Subsets bepalen (Oefenen)

Stel, de functie `subsets(xs)` geeft alle deelverzamelingen van `xs`. In de theorie heb je kunnen lezen hoe je zo'n functie recursief kunt maken.

Een vergelijkbare functie `choose(k, xs)` geeft alle deelverzamelingen van lengte `k`. Die zou je kunnen maken door eerst alle subsets op te sommen, en dan op basis van lengte te filteren. Maar dan reken je onnodig veel uit.

Je kunt het ook met een recursieve versie van `choose(k, xs)` die alleen zichzelf aanroeft. Geef deze functie.

Let op: je moet zowel nadenken hoe je een aanroep voor `choose` met een kleinere `k` kunt gebruiken als hoe je een aanroep met een kleinere `xs` kunt gebruiken.

## 4.12 Les 12: Recursie en Iteratie 2

### Tijdens de les

#### 12.1 Bestanden zoeken (Oefenen)

Je hebt gezien hoe je recursieve functies kunt toepassen op recursieve structuren zoals het bestands-systeem.

Schrijf nu de volgende Python functies:

- (a) `findfile(filename, path)` die zoekt naar een bestand met een specifieke naam in een map en al zijn submappen. De functie geeft een lijst van alle mappen waarin het bestand is gevonden terug.
- (b) `printfolder(path)` die een lijst met alle bestanden en mappen en submappen in een bepaalde map afdrukt. Voor iedere submap die je afdrukt moet je inspringen met een aantal spaties.

**TIP:** Gebruik de functie `os.listdir(path)` uit de `os` module om de inhoud van een map op te vragen.