Computertalen

(SV 1.4)

P.J. den Brok MA

12 februari 2013

Inleiding

De leerstof bestaat uit theoretische modellen afgewisseld en gekoppeld met toepassingen, voorbeelden en vaardigheidsoefeningen. Het een kan niet zonder het ander. Welke theoretische modellen worden behandeld?

- ⋆ De (on)mogelijkheden van reguliere talen.
- ⋆ De werking en realisaties van de eindige automaten DFA en NFA.
- ⋆ De realisatie van het patroonherkennen
- ⋆ De (on)mogelijkheden van contextvrije en contextgevoelige talen.
- ⋆ Het principe en de realisatie van een top-down en bottom-up parser.
- ⋆ De verschillen tussen diverse compilers (crosscompiling, pseudo-instructies) en interpreters (on-the-fly/just-in-time).
- ⋆ De mogelijkheden en beperkingen van macro's en templates.
- ⋆ De fasering en de modulaire opbouw van een compiler.
- ⋆ Het principe van de lexicografische analyse.
- ⋆ Het principe van de syntax analyse.
- ⋆ De functie van syntaxbomen.
- ⋆ De optimalisatie en codegeneratie met syntaxbomen.
- ⋆ De werking van stapelmachines.
- ⋆ De werking van registermachines.
- ⋆ De functionaliteit van een macro-assembler.
- ⋆ Realisatie van besturings-, data- en objecttructuren.
- ⋆ Virtuele machines, 'call-by-value', 'call-by-reference', recursie, statusvlaggen, Floating Point, interrupts en threading.

Welke toepassingen en vaardigheden zijn mogelijk na de bestudering en oefeningen?

- ⋆ Het opstellen van eenvoudige en geavanceerde reguliere expressies.
- ⋆ Het opstellen en toepassen van patroonherkenning voor beheer- en gebruiksproblemen, onderzoek van DNA-strings, tekstverwerking en bestandsconversie.
- ⋆ Het opstellen en interpreteren van grammatica's in (E)BNF voor protocollen, programmeer- en opmaaktalen
- ⋆ Het genereren van een scanner met '(f)lex' of 'Jlex'.
- ⋆ Het genereren van een parser met 'bison','yacc' of 'CUP'.
- ⋆ Het maken van een eenvoudige vertaler of interpreter.
- ⋆ Het ontwerpen en realiseren van programmatuur zoals: Foutdetectie en correctie in teksten en protocollen, plagiaatcontrole, XML-parsing en Data Mining en KI en daarvan de beperkingen kennen.
- ⋆ Het programmeren in assemblertaal in verband met snelheid- en ruimtebeperkingen in embedded systemen.

Inhoudsopgave

Hoofdstuk 1

Reguliere talen

Een taal bestaat uit woorden en zinnen. De woorden zijn samenstellingen van letters uit een alfabet, de zinnen zijn samenstellingen van woorden. Bij een gesproken taal zijn de bouwstenen geen letters maar klanken. Maar welke combinaties van bouwstenen leiden tot een echte taal? Het willekeurig aan elkaar plakken van letters of klanken zeker niet. De volgorde van de letters, de woorden en de zinnen worden bepaald door de betekenis, de 'semantiek', en door de regels die de vorm van de woorden en zinnen bepalen, de 'grammatica'.

Talen, waarvan de grammatica's volledig en exact beschreven kunnen worden, noemt men formele talen. De natuurlijke talen zoals Nederlands en Engels zijn te ingewikkeld om volledig formeel beschreven te worden. Omdat machines blijkbaar minder ingewikkeld zijn dan mensen, kunnen computertalen wel volledig formeel beschreven worden.

1.1 Reguliere talen

Om een reguliere taal te maken is een eindige verzameling 'tekens', ookwel 'alfabet' genoemd, noodzakelijk. Uit zo'n alfabet kunnen nieuwe combinaties gevormd worden. De verzameling A met bijvoorbeeld de tekens $\{a,b,c\}$ is zo'n een alfabet. Uit de tekens kunnen rijen samengesteld kunnen worden zoals: $u = aaaabcc$ en $v = ccaaa$. De rijen u en v worden 'strings' genoemd. Om deze strings wat korter te noteren wordt een algebraïsche notatie gebruikt: $u = aaaabcc = a^4bc^2$ en $v = ccaaa = c^2a^3$. In tegenstelling met het vermenigvuldigen in de normale algebra, is volgorde van de tekens in een string kenmerkend: $a^3b^2 \neq b^2a^3$. Deze eigenschap wordt 'niet-commutatief' genoemd.

De lengte van een string wordt aangegeven met $|u| = |a^4bc^2| = 7$. Dit is het aantal letters of de som van de exponenten in de algebraïsche notatie van het woord. Voor het gemak worden de strings aangegeven met $u, v, w, \ldots$ en de letters uit het alfabet met $a, b, c, \ldots$. Er is ook sprake van een 'lege string' ε, de string zonder lengte: $|\varepsilon| = 0$.

Een niet-lege 'deelstring' van de string $u = a_1a_2a_3\ldots a_n$ is een string $a_i\ldots a_k$ waarbij $1 \leq i \leq k \leq n$. De lege string ε is een deelstring van alle strings.

Sommige deelstrings hebben bijzondere namen. Bijvoorbeeld de 'prefix' van $u = a_1a_2a_3\ldots a_n$ is een deelstring die begint met $a_1\ldots a_j$. De 'suffix' van $u = a_1a_2a_3\ldots a_n$ is een deelstring die eindigt met a_n zoals $a_j\ldots a_n$. Een deelstring die geen prefix of suffix is, wordt een 'infix' genoemd.

Voorbeeld 1.1 *Van $w = abc$ zijn de prefixen: ε, a, ab en abc. De suffixen van $w = abc$ zijn ε, c, bc en abc. Alle deelstrings van $w = abc$ zijn ε, a, ab, abc, bc, c en b. Alleen ε en b zijn infixen.*

Strings kunnen aan elkaar geplakt worden, dit wordt een 'samenvoeging' genoemd. Als bijvoorbeeld $u = aaaabcc = a^4bc^2$ wordt samengevoegd met $v = ccaaa = c^2a^3$ dan is het resultaat $uv = aaaabccccaaa = a^4bc^4a^3$. Een samenvoeging is niet-commutatief, het is afhankelijk van de volgorde: $uv = a^4bc^4a^3 \neq vu = c^2a^7bc^2$. De lege string ε mag echter overal aan en tussen geplaatst worden: $\varepsilon ab = a\varepsilon b = ab\varepsilon = ab$.

Uit een alfabet A kan een oneindige verzameling strings ontstaan, inclusief de lege string ε. Deze verzameling strings geeft men aan met $A^\star$. Het asterix-symbool $(\star)$ betekent in deze notatie: 'nul of meer'. Er is ook een 'één of meer' notatie, het 'plus-symbool: $(+)$. De verzameling A^+ bestaat alle niet-lege-strings, die uit het alfabet A kunnen ontstaan.

Definitie 1.1 *Een taal L is een deelverzameling van alle strings $A^\star$, die uit een eindig alfabet A kunnen ontstaan.*

1.1.1 Eindige en oneindige reguliere talen

Van de talen met oneindig veel combinatie mogelijkheden, is de groep 'reguliere talen' het eenvoudigst. Alleen de talen met een eindig aantal combinaties zijn eenvoudiger. Voor het gemak worden ook deze eindige talen ook 'regulier' genoemd.

De definitie van een 'reguliere taal' L komt eigenlijk meer overeen met het begrip 'woordenschat' dan met het gangbare begrip 'taal'. Een taal is een samenstelling van woorden en zinnen, een woordenschat is maar een deelverzameling van alle mogelijke strings. Uit de woordenschat van een natuurlijke taal kunnen natuurlijk wel zinnen gevormd worden. In een formele taal is een zin een 'meta-woord', een woord van woorden. Onderscheid maken tussen woorden en zinnen is met deze formele definitie van 'taal' niet meer nodig.

De woordenschatten kunnen eindig en oneindig veel woorden bevatten. Een 'eindige reguliere taal' bevat een eindig aantal woorden, een 'oneindige reguliere taal' bevat een oneindig aantal woorden.

Voorbeeld 1.2

1. *De woorden a en bbb vormen de eindige taal* $L = \{a, b^3\}$*;*
2. *De woorden met even aantal a's: vormen de oneindige taal* $L = \{a^2, a^4, a^6 \ldots\}$*;*
3. *De woorden met één a en gevolgd door nul of meer b's vormen de oneindige taal* $L = \{a, ab, ab^2, \ldots ab^n\}$*;*
4. *woorden met één of meer a's gevolgd door één of meer b's vormen de oneindige taal* $L = \{a^m b^n; 0 < m, 0 < n\}$*;*
5. *De woorden met exact één a vormen de oneindige taal* $L = \{b^m a b^n; 0 \leq m, 0 \leq n\}$.

1.1.2 De samenvoeging van de woorden reguliere talen

Reguliere talen kunnen op woordniveau samengevoegd worden. Dit heet 'concatenatie' van woorden. Deze concatenatie geeft woordcombinaties, waarvan de prefix uit de ene- en de postfix uit de andere taal komt. Net zoals bij strings, is de concatinatie van de woorden niet-commutatief: $L_1 L_2 \neq L_2 L_2$. De resulterende woordenschat van $L_1 = \{a, b^2\}$ en $L_2\{a^2, ab, b^3\}$ staat in de tabel:

$L_1 \backslash L_2$	a^2	ab	b^3
a	a^3	$a^2 b$	ab^3
b^2	$b^2 a^2$	$b^2 ab$	b^5

De elementen in de tabel vormen de samengevoegde taal:

$$L_1 L_2 = \{uv; u \in L, v \in L_2\} = \{a^3, a^2 b, ab^3, b^2 a^2, b^2 ab, b^5\}$$

Op dezelfde manier kunnen een eindige $L_1 = \{a, b^2\}$ en de oneindige taal $L_3 = \{a^2, a^4, a^6 \ldots\}$, samengevoegd een oneindige taal vormen:

$$L_1 L_3 = \{uv; u \in L_1, v \in L_3\} = \{a^3, a^5, a^7, \ldots, b^2 a^2, b^2 a^4, b^2 a^6 \ldots\}$$

De samenvoeging van een taal $L = \{a, b^2\}$ met zichzelf wordt: $LL = L^2 = \{a^2, ab^2, b^2 a, b^4\}$.

$L \backslash L$	a	b^2
a	a^2	ab^2
b^2	$b^2 a$	b^4

De taal die alleen uit de lege string ε bestaat, wordt als $L^0 = \{\varepsilon\}$ genoteerd. Deze taal L^0, met de lege-string als woordenschat, is niet gelijk aan de 'lege taal' $L = \{\} = \emptyset$, de taal zonder woordenschat.

1.1.3 De vereniging van reguliere talen

Talen kunnen verenigd worden. Het resultaat is de vereniging van de twee woordenschatten L_1 en L_2:

$$L_1 \cup L_2 = \{a, b^2\} \cup \{a^2, ab, b^3\} = \{a, a^2, ab, b^2, b^3\}$$

Vereniging van talen is op woordenschatniveau. Daarentegen is samenvoeging van talen op woordniveau.

1.1.4 De afsluiting van reguliere talen

De 'niet-lege afsluiting' L^+ is de verzameling met oneindig veel niet-lege combinaties die door een eindige of oneindige woordenschat L gegenereerd kan worden.

$$L^0 = \{\varepsilon\} \quad L^1 = L \quad L^2 = LL \quad \ldots \quad L^n = LL^{n-1}$$

$$L^+ = L^1 \cup L^2 \cup L^3 \cup \ldots = \bigcup_{k=0}^{\infty} L^k$$

Als de lege string in L^+ wordt opgenomen, dan wordt dit de 'Kleene-afsluiting' $L^\star$ van de taal L genoemd.

Definitie 1.2

$$L^\star = L^0 \cup L^+ = \{\varepsilon\} \cup L^+$$

Voorbeeld 1.3 *Uit het alfabet A en de taal:*

- $L = \{a^2\}$ *volgt* $L^\star = \{$*alle even woorden* $a^{2n}; 0 \leq n\}$*;*
- $L = \{a, b\}$ *volgt* $L^\star = \{$*alle woorden*$\} = A^\star$*;*
- $L = \{a, b, c^3\}$ *volgt* $L^\star = \{$*alle woorden met de deelstring* $c^{3n}; 0 \leq n\}$.

De Kleene-afsluiting kan van een eindige taal L een oneindige taal $L^\star$ maken, dus een taal met een oneindig aantal combinatiemogelijkheden, inclusief de lege string.

1.1.5 Het verschil en complement van reguliere talen

Het verschil tussen twee talen is het verschil tussen de twee woordverzamelingen.

Voorbeeld 1.4 *Gegeven:* $L_1 = \{a^{2n}; 0 \leq n\}$ *en* $L_2 = \{\varepsilon, a, a^2, a^3, a^4\}$. *De taal* $L_1 - L_2 = \{a^6, a^8 \ldots\} = \{a^{2n}; 3 \leq n\}$. *De taal* $L_2 - L_1 = \{a, a^3\}$. *Het verschil tussen twee talen is dus niet-commutatief:* $(L_1 - L_2) \neq (L_2 - L_1)$.

De complementaire taal $\overline{L}$ is te definieren als alle strings uit het alfabet $A^\star$ die niet in L aanwezig zijn:

Definitie 1.3

$$\overline{L} = A^\star - L$$

Voorbeeld 1.5 *Gegeven de taal* $L = \{a^{2n}; 0 \leq n\}$. *De complementaire taal* $\overline{L} = \{a^{2n+1}; 0 \leq n\}$.

1.2 Reguliere talen en expressies

Een 'formele grammatica' beschrijft een taal L als een deelverzameling van alle strings die uit een alfabet A kunnen ontstaan. Als deze beschrijving mogelijk is met behulp van een expressie r, eventueel gecombineerd met de asterix/plus notatie, dan wordt deze formele grammatica een 'reguliere expressie' genoemd.

Definitie 1.4 *De taal* $L(r)$ *is regulier*[1] *als er een expressie r is, die de reguliere woorden beschrijft die uit een alfabet A kunnen ontstaan. Anderzijds, als een taal regulier is, dan is er ook een reguliere expressie. Voor reguliere talen en reguliere expressies gelden de volgende regels:*

1. *Voor elke letter a in het alfabet A geldt dat* $r = a$ *een reguliere expressie is. De (eindige) taal die bij deze reguliere expressie hoort is* $L(r) = L(a) = \{a\}$*;*
2. *Als r een reguliere expressie is, dan is ook* (r) *een reguliere expressie. De uitdrukking* (r) *komt overeen met* $L(r)$*. De haakjes worden - conform de rekenkunde - gebruikt te om delen van de expressie naar prioriteit te groeperen;*
3. *De lege string* ε *is een reguliere expressie. De taal die daar bij hoort is* $L(\varepsilon) = \{\varepsilon\}$*. Samenvoeging van de taal* $L(\varepsilon)$ *met een andere taal* $L(r)$ *verandert niets aan de resulterende woordenschat* $L(r) = L(r)L(\varepsilon) = L(\varepsilon)L(r)$*;*
4. *Een lege reguliere expressie* $r = ()$ *komt overeen met de lege taal* $L() = \{\} = \emptyset$*. Een samenvoeging van de lege taal* $L()$ *met een andere taal* $L(r)$ *elimineert de woordenschat* $L(r)L() = L()L(r) = \emptyset$*;*

[1] 'Regulier' betekent 'volgens de regels'.

5. *De symbolen* $(\star)$ *en* $(+)$ *staan niet voor de operatoren 'vermenigvuldigen' en 'optellen', maar zijn kwantoren bekend uit de afsluitingen van woordenschatten. De reguliere expressies* $r^\star$ *en* r^+ *komen overeen met resp.* $(L(r))^\star$ *en* $(L(r))^+$. *De woordenschatten die deze reguliere expressies opleveren, zijn oneindig groot:* $a^\star = (L(a))^\star = \{\varepsilon, a, a^2, a^3, \ldots\}$ *en* $a^+ = (L(a))^+ = \{a, a^2, a^3, \ldots\}$;

6. *De reguliere expressie* $r_1|r_2$ *betekent dat* r_1 *en* r_2 *alternatieven zijn. Reguliere expressies met alternatieven stellen de vereniging van woordenschatten voor* $L(r_1|r_2) = L(r_1) \cup L(r_2)$. *Alternatieven zijn commutatief:* $r_1|r_2 = r_2|r_1$ *en associatief:* $(r_1|r_2)|r_3 = r_1|(r_2|r_3)$;

7. *Als* r_1 *en* r_2 *geldige reguliere expressies zijn, dan is ook* $r_1 r_2$ *een geldige reguliere expressie. Deze constructie komt overeen met de samenvoeging van reguliere talen* $L(r_1 r_2) = L(r_1)L(r_2)$. *In tegenstelling met alternatieven, is een samenvoeging niet-commutatief:* $r_1 r_2 \neq r_2 r_1$. *De lege string* ε *daarentegen, mag overal voor-, achter- en tussengeplakt worden:* $r = \varepsilon r \varepsilon$. *Een samenvoeging is distributief over alternatieven:* $r_1(r_2|r_3) = (r_1 r_2)(r_1|r_3)$ *en* $(r_1|r_2)r_3 = (r_1 r_3)|(r_2|r_3)$.

Voorbeeld 1.6 *Voor een alfabet* $A = \{a, b\}$ *geldt dat:*

1. *de expressie* $r = a^\star$ *regulier is,* $L(r) = \{\varepsilon, a, a^2, a^3, \ldots\}$;
2. *de expressie* $r = a|b^\star$ *regulier is,* $L(r) = \{a, \varepsilon, b, b^2, b^3, \ldots\}$;
3. *de expressie* $r = (a|b)^\star$ *regulier is,*
   $(a|b)^\star = L(L(a) \cup L(b))^\star = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots, babb \ldots\}$;
4. *de expressie* $r = (a^\star|b^\star)$ *regulier is,*
   $(a^\star|b^\star) = L(a^\star) \cup L(b^\star) = \{\varepsilon, a, aa, aaa, aaaa, \ldots, b, bb, bbb, bbbb, \ldots\}$;
5. *de expressie* $r = a|^\star$ *niet-regulier is volgens de regel voor alternatieven;*
6. *de taal* $L_1 = \{a, b^2\}$ *regulier is,* $r = a|bb$;
7. *de taal* $L_2 = \{a^2, ab, b^3\}$ *regulier is,* $r = aa|ab|bbb$;
8. *de taal* $L_3 = \{\varepsilon, a^2, a^4, a^6 \ldots\}$ *regulier is,* $r = (aa)^\star$
9. *de taal* $L_4 = \{a, ab, ab^2, \ldots ab^n\}$ *regulier is,* $r = ab^\star$;
10. *de taal* $L_5 = \{a^m b^n; 0 < m, 0 < n\}$ *regulier is,* $r = a^+ b^+$;
11. *de taal* $L_6 = \{a^n b^n; 0 < n\}$ *niet regulier is;*
12. *de taal* $L_7 = \{b^m a b^n; 0 \leq m, 0 \leq n\}$ *regulier is,* $r = b^\star a b^\star$.
13. *de taal* $L_8 = \{c\} = \emptyset$ *leeg is* $(c \notin A)$ *en dus regulier is.*

1.3 De deterministische-eindige automaat (DFA)

Een deterministische eindige automaat (DFA), bestaat uit de volgende delen:

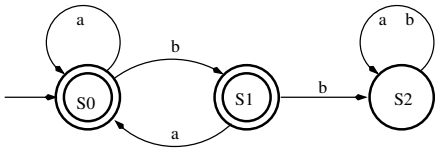
1. een verzameling invoertekens uit een eindig alfabet $A = \{a_1, a_2, \ldots a_m\}$;
2. een eindige verzameling toestanden $S = \{s_1, s_2, \ldots s_n\}$;
3. een (deel)verzameling eindtoestanden $Y \subset S$;
4. een begintoestand $s_0 \in S$;
5. een verzameling overgangen $F : S \times A \rightarrow S$, overgangen f_{ij} van s_i naar s_j zijn afhankelijk van invoertekens a_k.

De functie van een DFA is het accepteren of afwijzen van een een rij invoertekens. De DFA begint in de begintoestand s_0. Elk teken veroorzaakt een toestandsovergang van toestand s_i naar toestand s_j. Zodra de rij eindigt in een van de eindtoestanden, is de string geaccepteerd. Indien de rij eindigt in een andere toestand dan wordt de rij niet geaccepteerd.

Voorbeeld 1.7 *Een DFA met het alfabet $A = \{a, b\}$, $S = \{s_0, s_1, s_2\}$, begintoestand s_0 en eindtoestanden s_0 en s_1. De overgangen F zijn in de volgende tabel aangegeven:*

F	a	b
(s_0)	s_0	s_1
(s_1)	s_0	s_2
s_2	s_2	s_2

De eindtoestanden s_0 en s_1 zijn met haakjes aangegeven, de begintoestand is s_0. Omdat een tabel moeilijker te doorgronden is, kan men beter de DFA als een graaf tekenen. De toestanden zijn dan ballen, de overgangen zijn de pijlen. De eindtoestanden zijn met dubbele ballen aangegeven:

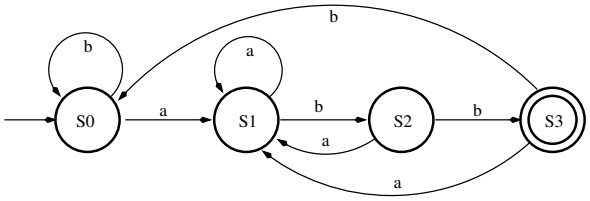


Figuur 1.1: DFA voor $r = (a|ba)^\star(\varepsilon|b)$

De taal die door deze DFA geaccepteerd wordt, zijn alle woorden waarin geen b^2 in voorkomen. Dit komt overeen met de reguliere expressie $r = (a|ba)^\star(\varepsilon|b) = ((a|ba)^\star)|(a|ba)^\star b)$. De lege string ε wordt door deze DFA geaccepteerd omdat de begintoestand s_0 tevens eindtoestand is.

Stelling 1.1 (Kleene) *Elke reguliere taal definieert een DFA en elke DFA definieert een reguliere taal.*

Voorbeeld 1.8 *Een DFA met het alfabet $A = \{a,b\}$, $S = \{s_0, s_1, s_2, s_3\}$ kan als een graaf (fig. 1.2) worden getekend.*



Figuur 1.2: DFA voor $r = (a|b)^\star ab^2$

De taal die door deze DFA gedefinieerd wordt, is de reguliere taal $L((a|b)^\star ab^2)$. De lege string ε *wordt niet geaccepteerd omdat in deze DFA de begintoestand s_0 geen eindtoestand is.*

1.3.1 De pompstelling

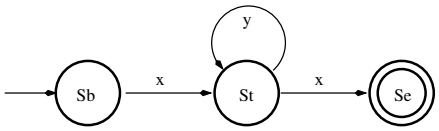
Neem aan dat er een DFA is met k toestanden en stel dat er een woord $w = a_1 a_2 \ldots a_n$ is zodat de lengte van dit woord groter is dan het aantal toestanden van de DFA: $k < |w| = n$. Omdat elk teken een toestandsovergang veroorzaakt, is er een bepaalde reeks waarin de toestanden van de DFA elkaar opvolgen, nummer deze reeks toestanden met $1 \ldots n$. Om deze nummering niet te verwarren met het identificatienummers van de toestanden: $s_0, s_1, \ldots s_k$ wordt de begintoestand aangegeven met s_b, de tussentoestanden s_t en de eindtoestand met s_e:

$$(s_b)_1, (s_t)_2, (s_t)_3, \ldots, (s_e)_n$$

Omdat k kleiner dan n is, moet er een tussentoestand s_t zijn met $0 < t \leq k-1$, die herhaald optreedt in de reekstoestanden: $(s_x)_i = (s_x)_j$ met $1 \leq i \leq j \leq n$. Dit is verklaarbaar met het duiventilprincipe dat stelt: "als er meer duiven dan tillen zijn, dan zijn er tillen met meer duiven".

$$x = a_1 a_2 \ldots a_i \qquad y = a_{i+1} \ldots a_j \qquad z = a_{j+1} \ldots a_n$$

Als x en y in dezelfde tussentoestand s_t eindigen, dan zal de ook de string xy eindigen in $(s_t)_i = (s_t)_j$. Hieruit volgt dat xy^m. Alle strings $w_m = xy^m z$ eindigen uiteindelijk in een eindtoestand $(s_{eind})_n$. Deze situatie wordt schematisch weergegeven in figuur 1.3, waarin de deelstrings x, y, z en de toestanden s_b, s_t, s_e symbolisch zijn aangegeven:



Figuur 1.3: Het pompschema (dit is géén DFA graaf)

Stelling 1.2 (De pompstelling voor reguliere talen) *Als een DFA met k toestanden, de strings x, y en z waarvoor geldt dat $k < |xyz|$ en $0 < |y|$ accepteert, dan zal de taal die door deze DFA gedefinieerd wordt, de woorden $xy^m z; 0 < m$ bevatten.*

Gegeven het woord $k < |a^m b^m|$. Neem aan dat dit woord geschreven kan worden als xyz, een samenvoeging van de deelstringen x y en z. Omdat y niet leeg is, dan kan y alleen a's óf alleen b's óf a's en b's samen bevatten. Volgens de pompstelling moet een DFA dan ook xy^2z accepteren. Als $a^m b^m = x(y)z = a^{m-i}(a^i b^j)b^{m-j}$ dan is $x(y)^2 z = a^{m-i}(a^i b^j)^2 b^{m-j} \neq a^n b^n$. Hieruit volgt dat de taal het woord $a^n b^n$ niet door de DFA met een eindig aantal toestanden $k < n$ geaccepteerd kan worden en de taal $L = \{a^n b^n; 0 < n\}$ volgens de stelling 1.1 niet regulier is.

Voorbeeld 1.9 *Als een DFA met $k = 7$, het woord $7 < |a^4 b^4|$ accepteert, geldt dan $xy^2z = a^n b^n$ voor een bepaalde n?*

Neem bijvoorbeeld $y = a^{i=2} b^{j=0}$:

$$\underbrace{aa}_{x}\overbrace{aa}^{y}\underbrace{bbbb}_{z} \quad \rightarrow \quad \underbrace{aa}_{x}\overbrace{aaaa}^{y^2}\underbrace{bbbb}_{z} \neq a^n b^n$$

of neem bijvoorbeeld $y = a^{i=2} b^{j=2}$

$$\underbrace{aa}_{x}\overbrace{aabb}^{y}\underbrace{bb}_{z} \quad \rightarrow \quad \underbrace{aa}_{x}\overbrace{aabbaabb}^{y^2}\underbrace{bb}_{z} \neq a^n b^n$$

Hoewel de pompstelling aan kan tonen dat een taal niet regulier is, kan zij niet aantonen dat een taal wel regulier is. De pompstelling kan ook niet gebruikt worden voor strings $|xyz| < k$.

Voorbeeld 1.10

1. *$L = \{a^{f(n)}\}; n \leq 0\}$ is regulier voor elke geheeltallige functie $f(n) = p \cdot n + q$ met constanten p en q. Er zijn dan p toestanden in de lus en q seriële toestanden;*
2. *$L = \{a^n b^m; 0 < n, 0 < m\}$ is regulier omdat er geen relatie tussen n en m is. Deze taal is gelijk aan $L(a^+ b^+)$;*
3. *$L = \{a^n b^n; 0 < n\}$ is niet regulier volgens de pompstelling;*

4. *$L = \{a^{f(n)}b^{f(n)}; 0 < n\}$ is niet regulier voor elke geheeltallige functie $f(n) = p \cdot n + q$ met constanten p en q. Voor $p = 1$ en $q = 1$ is dit hetzelfde als $L = \{aa^n bb^n; 0 < n\}$;*

5. *$L = \{a^n b^k; 0 < n < k\}$ is niet regulier omdat het niet mogelijk is om $n < k$ te controleren.*

1.3.2 De realisatie van een DFA

Een van de belangrijkste toepassingen voor reguliere expressies is patroonherkenning. Patroonherkenning past men toe om delen van strings te zoeken of te vervangen. Deze strings zijn niet altijd één-dimensionaal. Er bestaan ook seriële en parallelle patroonherkenners in twee- of meer-dimensionale structuren. Omdat patroonherkenning vaak grote hoeveelheden data betreft, zoals bijvoorbeeld DNA-strings met miljarden tekens, is de verwerkingssnelheid van het patroonherkennen zeer belangrijk.

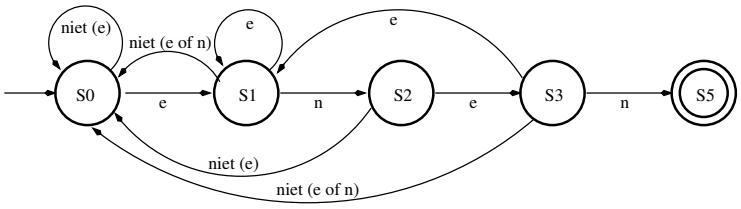
Voor eenvoudige reguliere expressies is een DFA aan te bevelen. Omdat een DFA tijdens patroonherkenning één voor één de invoertekens inleest en vergelijkt met de huidige toestand om naar een nieuwe toestand over te gaan, is de verwerkingssnelheid van een DFA meestal van de lineaire orde $O(n)$ (n het aantal tekens in de invoerstring). Naast de DFA-tabel zijn ook andere DFA-realisaties mogelijk:

Het 'brute-force-algoritme': Dit algoritme zoekt een patroon met m tekens in een invoerstroom met n tekens, door in $n - m + 1$ posities van de invoerstroom m tekens met een schuivend venster het patroon te zoeken. Bij een 'mismatch' springt het verder naar de volgende positie in de string. Bijvoorbeeld bij het zoeken van het patroon "enen" $m = 4$ in de string "eneenen" $n = 7$ maakt het 'brute-force-algoritme' gebruik van een schuivend venster van 4 karakters:

1. eneenen (4 vergelijkingen)
2. eneenen (4 vergelijkingen)
3. eneenen (4 vergelijkingen)
4. eneenen (4 vergelijkingen)

Totaal zijn dit 16 vergelijkingen. Op deze manier worden $m \cdot (n - m + 1) = n \cdot m - m^2 + m$ vergelijkingen gedaan. Nu met een DFA:

1. eneenen (1 vergelijking)
2. eneenen (1 vergelijking)
3. eneenen (1 vergelijking)
4. eneenen (1 vergelijking)
5. eneenen (1 vergelijking)

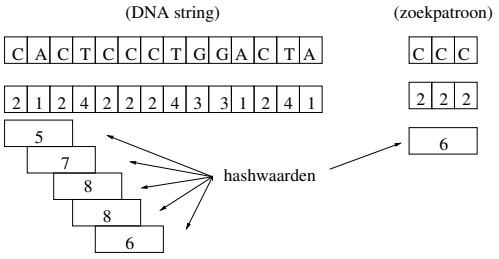


Figuur 1.4: DFA voor het patroon “enen”

6. eneenen (1 vergelijking)
7. eneenen (1 vergelijking)

Met de DFA is het aantal vergelijkingen maximaal het aantal tekens n in de invoerstroom $O(n)$;

Het ‘Karp-Rabin-algoritme’: Een verbetering van het ‘brute-force-algoritme’ is het algoritme van ‘Karp-Rabin’. Dit algoritme gebruikt hashwaarden van de string in plaats van de string zelf (fig. 1.5).



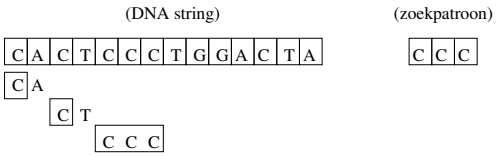
Figuur 1.5: Het ‘Karp-Rabin-algoritme’

De hashfunctie is in dit voorbeeld de modulo-16 som van de waarde w van de tekens $A = 1, C = 2, G = 3, T = 4$. Het algoritme begint met berekenen van de hashwaarde h van het patroon met m tekens en van de eerste m tekens van de string. Om de prestatie te verhogen wordt de hashwaarde van de volgende positie in de string berekend uit de hashwaarde *hash* van de huidige positie i in de string:

$$\begin{aligned} zoekhash &= patroon_1 + \ldots + patroon_m \quad modulo\ 16 \\ hash_1 &= teken_1 + \ldots + teken_m \quad modulo\ 16 \\ hash_{i+1} &= hash_i - teken_i + teken_{i+m} \quad modulo\ 16 \end{aligned}$$

Het gemiddelde verwerkingsnelheid van dit algoritme is van de orde $O(n+m)$. Hoewel dit algoritme relatief snel is, treden er soms collisions op als de verschillende deelstrings dezelfde hashwaarde hebben. Dan moeten de deelstrings op een andere manier gecontroleerd worden. Theoretisch wordt door de collisions de meest ongunstige verwerkingssnelheid $O(nm)$ bepaald;

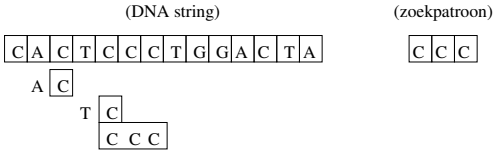
Het 'Knuth-Morris-Pratt-algoritme': Dit algoritme komt eigenlijk het meest overeen met het DFA-model. Bij het niet-herkennen van een patroon vanuit een bepaalde positie wordt er niet zondermeer naar de volgende positie gestapt. Het algoritme slaat alle posities over die geen prefix kunnen zijn van het patroon. Daarbij wordt gebruik gemaakt van de informatie die verkregen is bij het vergelijken van het patroon. Vervolgens wordt met behulp van een DFA-tabel de nieuwe positie bepaald.



Figuur 1.6: Het 'Knuth-Morris-Pratt-algoritme'

Het opstellen van de tabel heeft een tijdgedrag $O(m)$. Het 'Knuth-Morris-Pratt-algoritme' gedraagt zich in het slechtste geval als een $O(n+m)$ algoritme. Het algoritme is niet sneller dan het 'brute-force-algoritme' en het 'Karp-Rabin-algoritme'. Het voordeel van dit algoritme is dat er niet wordt terug gestapt naar de vorige posities in de invoerstring, het 'backtracken'. Dit kan bij 'real time streaming' een voordeel zijn;

Het 'Boyer-Moore-algoritme': Het algoritme van 'Boyer-Moore' lijkt op het 'Knuth-Morris-Pratt-algoritme', maar kan alleen worden toegepast als backtracking geen problemen geeft. In tegenstelling met de prefixen van het 'Knuth-Morris-Pratt', wordt in dit algoritme met suffixen gewerkt. Ondanks dat het 'Boyer-Moore-algoritme' - zoals gebruikelijk is - links in de invoerstring begint, wordt het patroon van achteren naar voren vergeleken. Als het patroon niet overeen komt met de laatste letter, die het eerst aanbod komt, wordt het in zijn geheel m posities naar rechts in de invoerstring geschoven. Komt een gedeelte s van de suffix overeen dan wordt het patroon maar $m-s$ posities naar rechtsgeschoven.



Figuur 1.7: Het 'Boyer-Moore-algoritme'

Als er bijna geen overeenkomstige suffixen zijn, dan heeft het algoritme een verwerkingssnelheid van de orde $O(n/m)$. Het 'Boyer-Moore-algoritme' werkt niet zo goed bij binaire strings omdat er veel overeenkomstige suffixen optreden. Soms is het mogelijk dit te verhelpen door de bits te groeperen in bytes of integers.

In het algemeen hebben de bovengenoemde algoritmen een verwerkingssnelheid die vergelijkbaar is met de DFA. Maar de prestatieverschillen worden veroorzaakt door de omstandigheden waarin deze algoritmen worden toegepast. Kleine verschillen tussen de algoritmen kunnen bij omvangrijke hoeveelheden gegevens wel leiden tot grote verschillen in de totale zoektijd.

1.4 De niet-deterministische-eindige automaat (NFA)

Het construeren van een DFA uit een reguliere expressie kan ingewikkeld zijn. Daarom maakt men gebruik van de niet-deterministische eindige automaat (NFA). Een NFA lijkt meer op een reguliere expressie dan een DFA. Uit elke NFA is uiteindelijk een DFA te construeren en omgekeerd. Het is wel mogelijk dat een DFA (soms exponentieel) meer toestanden heeft dan de overeenkomstige NFA. Voor een uitgebreide behandeling van $r \rightarrow NFA$ en het algoritme $NFA \rightarrow DFA$ wordt verwezen naar [1].

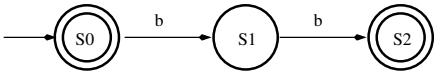
Een NFA verschilt op wezenlijke manieren van een DFA. Een NFA kan:

1. in elke toestand stoppen indien een invoerkarakter niet tot een overgang leidt;
2. gelijktijdig in meerdere toestanden verkeren;

De NFA heeft voor- en nadelen t.o.v een DFA.

1. Een NFA is gemakkelijker te begrijpen en te construeren dan een DFA;
2. Een DFA heeft vaak (exponentieel) meer toestanden dan een NFA;
3. De efficiëntie van een NFA wordt nadelig beïnvloedt door het backtracken. Een NFA moet na een verkeerde keuze terug naar een vorige toestand. Backtracking bij alternatieven met de zelfde prefix zoals $r = ab|ac$, kan vaak verholpen worden door factorisatie $r = a(b|c)$. Omdat een DFA geen backtracking heeft, is een DFA efficiënter dan een NFA.

Voorbeeld 1.11 *De taal die door de NFA in figuur 1.8 geaccepteerd wordt, is* $L(\varepsilon, b^2)$. *De lege string* ε *wordt geaccepteerd omdat de begintoestand een eindtoestand is.*

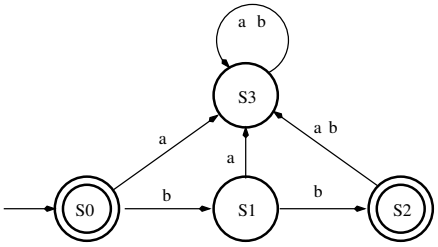


Figuur 1.8: Simpele NFA voor $r = \varepsilon|bb$

Om de tabel van een NFA te onderscheiden van een DFA worden de overgangen in plaats van de functiewaarden van F als transities T aangegeven.

T	a	b
(s_0)		s_1
s_1		s_2
(s_2)		

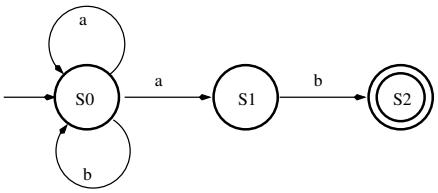
Men kan uit NFA in figuur 1.8 de volgende DFA afleiden:



Figuur 1.9: Simpele DFA voor $r = \varepsilon|bb$

De afgeleide DFA heeft meer toestanden, overgangen en een complexere structuur dan de oorspronkelijke NFA.

Voorbeeld 1.12 *De volgende NFA (fig. 1.10) accepteert de reguliere taal* $L = \{(a|b)^\star ab\}$.



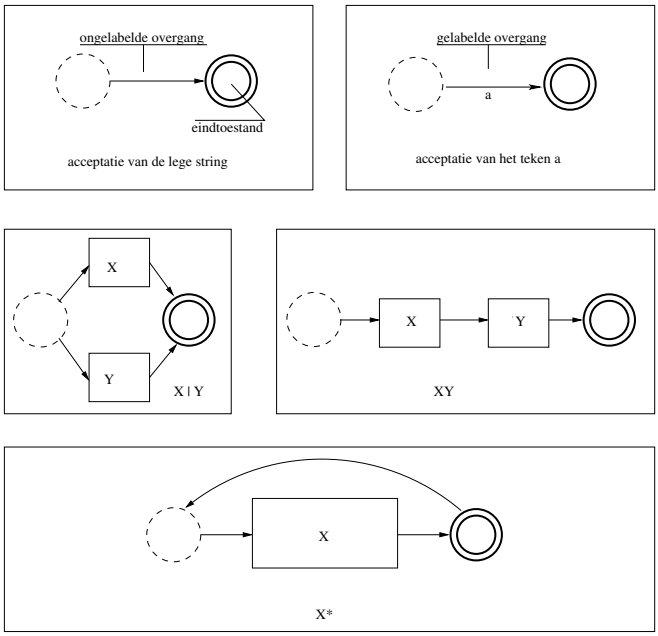
Figuur 1.10: NFA voor $r = (a|b)^\star ab$

T	a	b
s_0	$\{s_0, s_1\}$	s_0
s_1		s_2
(s_2)		

Merk op, dat vanuit toestand s_0 *na een a de NFA in twee toestanden verkeert.*

1.4.1 De constructie van een NFA

Omdat reguliere expressies op NFA's lijken, kan men bij het opstellen van schematische voorstellingen van NFA's rechtstreeks uitgaan van de reguliere expressie.

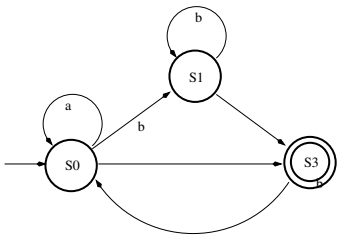


Figuur 1.11: NFA-constructies

Bij de constructie van een NFA gebruikt men een soort ‘legosteen’. Elke legosteen is een NFA en kan weer gebruikt worden in - en aan - andere legostenen. Op deze manier worden er wel veel lege-overgangen (de ongelabelde overgangen) en toestanden in het schema geïntroduceerd. Er zijn methodes om het aantal toestanden en het aantal lege-overgangen te reduceren [1].

Met de constructie van een NFA kan men aantonen of een taal regulier is. Want elke reguliere taal heeft een NFA en elke NFA definieert een reguliere taal (een indirect gevolg van de stelling van 1.1)

Voorbeeld 1.13 *De taal $L((a^\star b^+)^\star)$ is regulier omdat zij mogelijk is met de constructie in figuur 1.12.*



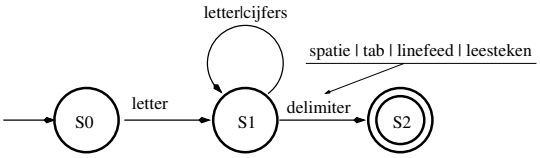
Figuur 1.12: Constructie voor $(a^\star b^+)^\star$

1.4.2 De realisatie van een NFA

Een van de belangrijkste criteria voor de keuze voor een realisatie van een DFA, is de complexiteit van de reguliere expressie. Bij zeer complexe reguliere expressies neemt het aantal toestanden en overgangen in een DFA meer dan lineair toe. Een NFA realisatie is dan, mits de zoektijden acceptabel blijven, te overwegen.

Een NFA wordt anders gerealiseerd. Elk teken in de reguliere expressie wordt vergeleken met een teken in de tekst. Het begin van het patroon in de tekst wordt vastgelegd met een wijzer. Indien het invoerteken niet herkend wordt als onderdeel van het patroon, gaat de wijzer naar een vorige positie in de tekst, het backtracken. Dit betekent dat een NFA posities moet bewaren. NFA's worden vaak als tabellen geïmplementeerd. Het niet-deterministische gedrag van een NFA kan tot zeer grote hoeveelheden tabellen leiden met bijna dezelfde inhoud. Het is mogelijk NFA's rechtstreeks in broncode te realiseren:

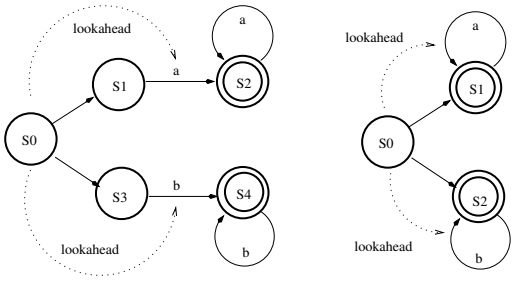
Voorbeeld 1.14 *Gegeven een eenvoudige NFA voor herkenning identifiers.*



Figuur 1.13: Herkenning van 'identifiers'

```
state0: save(pointer)                  /* tekstpositie op de stack  */
        c:=nextchr();                  /* lees invoerteken uit tekstbuffer */
        if letter(c) then goto state1;
        goto fail;                     /* geen herkenning */
state1: c:=nextchar();                 /* volgende invoerteken tekstbuffer */
        if letter(c) or digit(c) then goto state1;
        else if delimiter(c) then goto state2;
        goto fail();                   /* geen herkenning */
state2: restore(pointer);              /* tekstpositie van de stack */
        return("id");                  /* herkenning geslaagd */
fail:   restore(pointer);              /* tekstpositie van de stack */
        return("nill");                /* gefaald */
```

Alternatieven kunnen als ongelabelde (lege) overgangen in een NFA-schema worden weergegeven. De keuze voor een alternatief wordt dan afhankelijk gemaakt door toekomstige invoertekens (lookahead's). Dit wordt gerealiseerd door het voorlopig inlezen van invoertekens die verderop in de invoerstring staan met behulp van meerdere positiewijzers naar de invoerstring.



Figuur 1.14: NFA's met lege-overgangen voor $r = a^+|b^+$ en $r = a^\star|b^\star$

Als er sprake is van een reguliere expressie met kwantoren, dan zal een NFA zoveel mogelijk tekst proberen te herkennen. Indien dat niet lukt, valt de NFA terug op de vorige positie (backtracken). Een moderne realisatie van de NFA is de POSIX-NFA.

1.5 Opgaven

1. Geef van de volgende talen over $A = \{a,b,c\}$ de reguliere expressie r:

   (a) alle woorden met één a, gevolgd door nul of meer b's;

   (b) alle woorden met nul of meer a's, gevolgd door nul of meer b's of c's;

   (c) alle woorden met nul of meer a's of b's, gevolgd door één of meer abc's;

   (d) alle woorden met nul of meer abc gevolgd door één of meer a's of b's;

   (e) alle even woorden $a^{2n}; 0 \leq n$;

   (f) alle woorden;

   (g) alle woorden met de deelstring $c^{3n}; 0 \leq n$.

2. Wat is een 'zin' in een formele taal?

3. Zijn de volgende talen regulier?

   (a) $L = \{a^n b^m; 0 \leq n, 0 \leq m\}$

   (b) $L = \{a^{f(n)} b^{f(n)}; n \leq 0\}$ voor elke geheeltallige functie $f(n) = p \cdot n + q$ met de constanten $0 \leq p$ en $0 \leq q$;

   (c) $L = \{a^{3^n} a^2; 0 < n\}$;

   (d) $L = \{a^{3^n} b^2; 0 < n\}$.

4. Waarom kan de pompstelling nooit gebruikt worden om aan te tonen dat een taal regulier is? Hoe zou het dan wel moeten?

5. Vereenvoudig de reguliere expressies:

   (a) $r = (a)^\star$;

   (b) $r = (a^\star)^\star$;

   (c) $r = (\varepsilon|a^+|b^+)$;

   (d) $r = a^\star a^\star$;

   (e) $r = b^\star(ab^\star)^\star$.

   (f) $r = (a^\star|b^\star)^\star$.

6. Om een reguliere expressie te vinden, is het soms gemakkelijker om eerst een DFA (eventueel via een NFA) te construeren.

   (a) Construeer een DFA voor de taal $L = \{w \in \{a,b\}^\star; w$ bevat een even aantal a's en even aantal b's $\}$. (Hint: Ga uit van een starttoestand en vier toestanden: oneven a/even b, even a/oneven b, oneven a/oneven b en even a/even b);

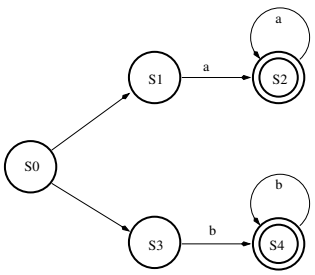
   (b) Construeer een reguliere expressie voor deze taal (Hint: Welke paden zijn er van begin- naar eindtoestand?).

   (c) Construeer de complementaire DFA voor de complementaire taal. Dit zijn alle woorden met geen even a's of geen even b's;

7. (a) Is elke DFA een NFA?
   (b) Is elke NFA een DFA?

8. Pas de reguliere expressie $r = abc^\star|ab^+$ aan, zodat er geen backtracking optreedt bij een NFA.

9. Bepaal van de volgende NFA in figuur 1.15 de overeenkomstige graaf van de DFA.



Figuur 1.15: nfa3

10. Welk algoritme zou u kiezen voor 'real time audio streaming' om patronen te herkennen: Het 'brute-force'-, het 'Karp-Rabin'-, het 'Knuth-Morris-Pratt'- of het 'Boyer-Moore-algoritme'? Verklaar uw antwoord.

11. In celkernen bevindt zich DNA (deoxyribonucleic acid), moleculen met lange ketens van basenparen. Elk basenpaar wordt aangegeven een van de volgende 4 letters: *A* ('adenine'), *T* ('thymine'), *C* ('cytosine') en *G* ('guanine'). Waarom werkt het 'Boyer-Moore-algoritme' niet optimaal met DNA-strings? Welke verbetering is mogelijk? Verklaar uw antwoord.

12. Geef een betere hashfunctie voor 'Karp-Rabin-algoritme' dan die in figuur 1.5. Verklaar uw keuze.

13. Wanneer zou een NFA toegepast worden i.p.v. een DFA? Hoe zou deze gerealiseerd kunnen worden?

Hoofdstuk 2

Patroonherkenning

De patroonherkenning is een belangrijk onderdeel van tekstverwerkende programma's. Om efficiënt teksten te zoeken of te vervangen, wordt gebruik gemaakt van reguliere expressies. Behalve in editors, compilers en browsers worden reguliere expressies ook gebruikt in andere gereedschappen zoals 'grep', 'sed' en '(g)awk' [15].

Bekende scriptingtalen zoals 'Perl' [17], 'Python' [18], 'PHP' [19] en 'TCL' [20] hebben ingebouwde interpreters voor patroonherkenning met reguliere expressies. Om in 'Java' rechtstreeks met reguliere expressies te werken is er de 'java.util.regex' class [22]. Voor 'C' en 'C++' is er de 'regex' library [23].

De voorbeelden die in deze lessen gegeven worden, zijn geschikt voor 'sed' en '(g)awk' (app. D). Omdat de NFA realisatie in 'Perl' extra mogelijkheden biedt, zijn de geavanceerde voorbeelden in deze taal gegeven. Soms is er gekozen voor een hybride notatie om het principe te duidelijker te verklaren.

Bijvoorbeeld, in de voorbeelden wordt een patroon begrensd door voorwaartse schuine strepen (/). Deze stijl wordt door zowel 'sed', '(g)awk' en 'Perl' gebruikt. Daarentegen worden groepshaakjes ((), ()) en de (+) kwantor in de Perl-stijl gegeven.

sed: `/[A-Z][A-Z]*\(abc|xyz\)*/`
Perl: `/[A-Z]+(abc|xyz)*/`

Een overzicht van de patroonherkenning met reguliere expressies is te vinden in app. B.

2.1 Eenvoudige patronen

Het eenvoudigste patroon is een opeenvolging van één of meer letters of cijfers. Met een patroon kan in een bestand gezocht worden naar een stukje tekst dat daarmee overeen komt. Bij het zoeken naar patronen wordt normaal onderscheid gemaakt tussen kleine letters en hoofdletters.

```
/d/
```

```
Merck toch hoe sterck
nu in 't werck sich al steld,
Die 't allen tijd so ons
vrijheit heeft bestreden.
```

Symbolen hebben een bijzondere betekenis in een reguliere expressie. Dit in tegenstelling met tekens, de karakters die letterlijk onderdeel zijn van het patroon. Hoewel normale karakters zoals letters en cijfers tekens zijn, kunnen zij een bijzondere betekenis krijgen door een backslash (\) ervoor te plaatsen. Dit plaatsen van een backslash om de betekenis van een karakter te veranderen, wordt 'escapen' genoemd. Anderzijds kunnen bijzondere karakters zoals (^) en ($) die al een bijzondere betekenis hebben, door een backslash gewone tekens worden.

```
/.*/
Bijzondere karakters moeten een backslash krijgen.*
```

```
/\.\*/
Bijzondere karakters moeten een backslash krijgen.*
```

Er zijn ook symbolen die worden voorafgegaan door een (\), zoals het bijvoorbeeld het symbool ($\backslash d$) voor een willekeurig cijfer.

2.1.1 Positiesymbolen

Twee bijzondere karakters worden in reguliere expressies gebruikt om het begin en het einde van een regel aan te geven: het beginregel-symbool (^) en het einderegel-symbool ($). Deze symbolen worden gebruikt om de positie van het patroon in een regel aan te geven.

```
/^Merck/
Merck toch hoe sterck
"Merck toch hoe sterck"
```

```
/erck$/
Merck toch hoe sterck
"Merck toch hoe sterck"
```

De lengte van het een positiesymbool is nul, daarom passen de positiesymbolen (^) en ($) bij patronen met de lengte 0. Bijvoorbeeld, een lege regel kan vertaald worden als het patroon "/^$/".

Wat is het verschil tussen ($) en een gebruikelijke regelafsluiter zoals het linefeed-karakter ($\backslash n$)? Het symbool ($) staat voor de positie van het einde van de regel en heeft de lengte 0. Het ($\backslash n$) teken heeft een lengte 1. In 'UNIX' wordt het 'linefeed'-karakter ($\backslash n$) gebruikt als regelafsluiter in teksten. In 'DOS' en 'MS-Windows' wordt de 'return'-karakter ($\backslash r$) gevolgd door een ($\backslash n$) gebruikt als regelafsluiter.

2.1.2 Wildcards

In reguliere expressies kan het symbool (.) elk teken voorstellen, het wordt daarom een 'wildcard' genoemd. Daarentegen wordt een gewone punt in het letterlijke gedeelte van een reguliere expressie met een "(\.)" aangegeven. Het wildcard-symbool (.) staat voor alle tekens behalve de regelafsluiters zoals ($\backslash n$). De meeste programma's bieden echter de mogelijkheid om de regelafsluiters door de wildcard te laten accepteren.

```
/.s/
Die 't allen tijd so ons
vrijheit heeft bestreden.
```

```
/\./
vrijheit heeft bestreden.
```

2.2 Samenstellingen van patronen

Een patroon kan uit tekens en symbolen bestaan. Patronen kan men met haakjes groeperen tot een grotere eenheden. Deze haakjes zijn symbolen die niet bij het letterlijke gedeelte van het patroon horen. Groepen markeren niet alleen delen van een patroon als eenheden, maar worden tijdens het patroonherkennen apart opgeslagen, zodat er later naar terugverwezen kan worden.

```
/(erck)( )(si)/
Merck toch hoe sterck
nu in 't werck sich al steld,
```

In oudere UNIX-programma's zoals 'sed' en 'grep', moeten groepshaakjes met backslash worden aangegeven "/\(erck\)/". In 'Perl' hebben de groepshaakjes geen backslash "/(erck)/" (zie app. C). Dat betekent wel dat letterlijke haakjes een backslash nodig hebben.

2.2.1 Verzamelingen

Op de plaats van één enkel teken, kan in een patroon sprake zijn van een aantal mogelijke tekens.

Zo'n verzameling tekens wordt met vierkante haken aangegeven, bijvoorbeeld: "`/[aeiou]/`". Dit is de expliciet aangegeven verzameling van de (kleine) klinkers. Een impliciete manier om een verzameling aan te geven is de eerste en de laatste teken van een opeenvolgende reeks tekens te combineren met het (-) symbool, bijvoorbeeld: "`/[B-K]/`". Dit zijn alle hoofdletters (B) t/m (K). Of met "`/[A-Za-z]/`" als de verzameling van alle hoofd- en kleine letters. Een impliciete of expliciete verzameling gedraagt zich als een wildcard voor zijn elementen.

```
/[a-z]s/
Die 't allen tijd so ons
vrijheit heeft bestreden.
```

Veel programma's bieden verzamelingen aan met behulp van een bijzonder symbool, zoals ($\backslash w$) voor de witte ruimte en ($\backslash d$) voor de cijfers. Dit kan een reguliere expressie compacter maken, zoals "`/\d\d/`" in plaats van "`/[0-9][0-9]/`".

Het symbool (^) kan twee verschillende betekenissen hebben. Soms betekent (^) hetzelfde als het begin van de regel zoals in "`/^Merck/`". Maar als (^) aan het begin van een verzameling tekens staat, dan staat het voor alles wat niet in de verzameling aanwezig is. Dus "`/[^aeiou]/`" is de verzameling van alles wat geen (kleine) klinker is.

```
/[^a-z]s/
Merck toch hoe sterck
nu in 't werck sich al steld.
Die 't allen tijd so ons
vrijheit heeft bestreden.
```

2.2.2 Alternatieven

Een verzameling tekens is te beschouwen als een aantal alternatieven voor één letter. Maar als men alternatieve patronen wil aangeven die uit meer tekens of symbolen bestaat, dan gebruikt men het ($|$) symbool.

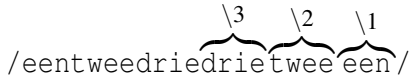
```
/kat|kattenvoer/
De kat at het kattenvoer.
```

Bij alternatieven patronen met dezelfde prefix wordt o.a. door de 'POSIX-NFA' de voorkeur gegeven aan het langste alternatief:

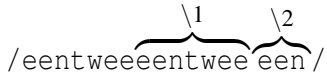
```
/kat|kattenvoer/
Waar is het kattenvoer?
```

2.2.3 Verwijzingen

Als men patronen nog een keer wil gebruiken, maar in een andere volgorde, dan kan men gebruik maken van terugverwijzingen. Men wijst terug naar een gegroepeerd patroon met behulp van één cijfer voorafgegaan door een backslash. Symbool $(\backslash 1)$ verwijst naar de eerste groep, symbool $(\backslash 2)$ naar de tweede groep etc. Op deze manier zijn 9 terugverwijzingen mogelijk. In 'Perl' mag men doorgaan met $(\backslash 10)$ en verder, mits dit aantal overeenkomt met het aantal patroongroepen. Hoewel de terugverwijzingen naar gegroepeerde patronen in volgorde overeenkomen, hoeven zij zelf niet in volgorde te staan. Bijvoorbeeld de constructie "`/(een)(twee)(drie)\3\2\1/`" komt overeen met:



Geneste groepen worden van buiten naar binnen geteld. De nummering van de terugverwijzingen is gekoppeld aan de nummering van open haakjes (). Bijvoorbeeld de constructie "`/((een)twee)\1\2/`" komt overeen met:



Terugverwijzingen naar patronen spelen een belangrijke rol bij het backtracken en het wijzigen van de teksten.

2.3 Kwantoren

Als men een patroon wil specificeren dat 'nul-of-meer' keer herhaald moet optreden, dan kan men de $(\star)$ kwantor direct achter het patroon plaatsen. Indien men 'een of meer' malen het patroon wil specificeren, dan plaats men de $(+)$ kwantor achter het patroon.

```
/A(CG)*A/
ACGTACGAATCGCATAATCGA
```

```
/A(CG)+A/
ACGTACGAATCGCATAATCGA
```

Als men een patroon wil specificeren dat 'nul-of-één' keer moet optreden, dan kan men de $(?)$ kwantor direct achter het patroon plaatsen ($a? = \varepsilon|a$). Zoals bijvoorbeeld bij de specificatie van een integergetal:

```
/[-+]?[0-9]+/
+2002 -99 +1.2E-222 3343 121.34 -
```

of om afkortingen van een woord te maken:

```
/a(f(k(o(r(t(i(ng?)?)?)?)?)?)?)?/
a,af,afk,afko,afkor,afkort,afkorti,afkortin en afkorting
```

Het is mogelijk om met de $(\star)$ kwantor de $(+)$ of $(?)$ kwantor te construeren. Bijvoorbeeld "`/(ACG)+/`" is gelijkwaardig met "`/(ACG)(ACG)*/`" en "`/T(ABC)?G/`" is gelijkwaardig met "`/TABCG|TG/`".

2.3.1 Numerieke kwantoren

Men kan een vast aantal patronen opgeven met behulp van een numerieke kwantor. Tussen $(\{)$ en $(\})$ kan men één of twee niet-negatieve gehele getallen plaatsen. Bijvoorbeeld, het zoeken in een tekstbestand naar regels met:

exact 80 karakters	`/^.{80}$/`
minstens 80 karakters	`/^.{80,}$/`
hoogstens 80 karakters	`/^.{,80}$/`
minstens 40 en hoogstens 80 karakters	`/^.{40,80}$/`

2.3.2 Gulzigheid

Kwantoren zijn in principe gulzig, zij blijven zoeken tot er niets meer van hun gading te vinden is. Soms blijkt een patroon deel uit te maken van groter patroon dat aan de zelfde reguliere expressie voldoet en vinden kwantoren meer dan bedoeld is. Daarom moet men voorzichtig zijn met patronen die gemaakt worden met de gulzige kwantoren $(\star)$ en $(+)$.

```
/d.*t/
aandachtig
dit patroon is te groot
```

De beste manier om gulzigheid te voorkomen is een patroon niet te formuleren met een stopvoorwaarde, maar met een geldigheidsvoorwaarde.

Vanaf "`d`" tot "`t`":

```
/d[^t]*/
aandachtig
dit patroon is te groot
```

Vanaf "`d`" tot en met "`t`":

```
/d[^t]*./
aandachtig
dit patroon is te groot
```

2.3.3 Backtracken

Terugverwijzingen gecombineerd met kwantoren kunnen in combinatie met backtracken bijzonder sluw toegepast worden. Bijvoorbeeld als de suffix van een woord overeenkomt met de prefix van het volgende woord, dan moeten de beide woorden vervangen worden door de grootste overeenkomstige deelstring:

```
echo "roofkatten kattenvoer"|perl -pe 's/([a-z]+)([a-z]+) \2([a-z]+)/\2/'
>katten

echo "roofkatten hondenvoer"|perl -pe 's/([a-z]+)([a-z]+) \2([a-z]+)/\2/'
>roofkatten hondenvoer
```

In het eerste voorbeeld stopt het backtracken zodra:

Door de gulzigheid van $(+)$ van de groep $(\backslash 1)$ wordt eerst het gehele woord "`roofkatten`" gevangen. Voor de groep $(\backslash 2)$ zijn er dan geen tekens meer over, het patroon kan niet herkend worden. Daarom zal een backtrackende NFA één karakter terugstappen en de (n) in "`roofkatten`" uit $(\backslash 1)$ overplaatsen naar $(\backslash 2)$. Dit zal zich herhalen totdat $(\backslash 2)$ overeen komt met "`katten`" en het patroon voldoet.

Dit geeft wel problemen met woorden waarvan de laatste letter van het eerste woord overeen komt met de eerste letter van het tweede woord (zie opgave 8).

2.4 Het wijzigen van patronen

Tekstverwerkende programma's combineren vaak het zoeken van teksten met het wijzigen van teksten. De tekstverwerkende programma's en scriptingtalen hebben vaak extra opties voor het wijzigen van tekst met behulp van reguliere expressies.

Belangrijk bij het wijzigen is zich te realiseren wat er gewijzigd wordt, of de wijziging ongedaan kan worden en hoever de reikwijdte van de wijziging is. Ook is het belangrijk te weten hoe er gewijzigd wordt. Tekstgeoriënteerde filter commando's in UNIX maken gebruik van pipes en redirection naar files, directe invoer en het scherm voor veranderingen van teksten (app. D). Andere tekstverwerkende programma's wijzigen teksten in geopende files. Het programma 'sed' daarentegen plaatst standaard de gewijzigde uitvoer standaard op het scherm en zal de invoerfile ongewijzigd laten.

De voorbeelden zijn gebaseerd op 'sed'. Het wijzigen werkt op een invoerstring, aangegeven door een $(<)$ gevolgd door invoer en een door uitvoer aangegeven met een $(>)$ gevolgd. Het bevel (s) om te wijzigen staat voor de eerste schuine streep, het zoekpatroon

staat tussen de eerste en de tweede schuine streep, de vervangende patroon staat tussen de tweede en de derde schuine streep. Bijvoorbeeld om overal "`dit`" te vervangen door "`dat`" wordt het commando "`s/dit/dat/g`" gegeven. Het argument (g) betekent dat het voor alle "`dit`" geldt en niet alleen voor de eerste.

```
s/honden/slangen/g
< gevaarlijke katten, wilde honden en zeehonden.
> gevaarlijke katten, wilde slangen en zeeslangen.
```

Tekstverwerkende programma's zouden dit soort wijzigingen zonder reguliere expressies kunnen uitvoeren. Maar zodra er alternatieven en kwantoren bij betrokken zijn, is de keuze voor reguliere expressies het meest praktisch.

```
s/honden|katten/slangen/g
< gevaarlijke katten, wilde honden en zeehonden.
> gevaarlijke slangen, wilde slangen en zeeslangen.
```

```
s/[a-z]+i[a-z]*[^ ]/lieve/g
< gevaarlijke katten, wilde honden en zeehonden.
> lieve katten, lieve honden en zeehonden.
```

Opmerking: 'sed' kent geen (+) symbool:

```
s/[a-z][a-z]*i[a-z]*[^ ]/lieve/g
< gevaarlijke katten, wilde honden en zeehonden.
> lieve katten, lieve honden en zeehonden.
```

Terugverwijzingen zijn niet alleen handig bij het opstellen van patronen, maar ook bij het vervangen van teksten. Bij een wijziging van de tekst waarin men een deeltekst van plaats moet verwisselen met een andere deeltekst, zijn terugverwijzingen de juiste oplossing.

```
echo "een twee"|sed -e 's/\(.*\) \(.*\)/\2 \1/'
> twee een
```

Er is reeds gewaarschuwd voor gulzige kwantoren. Dit gevaar is ernstiger wanneer er meer delen van de tekst gewijzigd worden.

2.4.1 Argumenten

Reguliere expressies kunnen extra argumenten krijgen om hun gedrag te wijzigen. Heel bekend is het (g) argument, dat aangeeft dat de reguliere expressie geldig is voor de gehele invoer. Sommige programma's zoals 'sed' vereisen dan ook dat dit argument wordt meegegeven aan de reguliere expressie. Indien dat niet gebeurt, dan stopt de reguliere expressie bij de eerste gevonden stukje tekst dat voldoet aan het patroon.

g - 'global': Zoek en vervang in de gehele tekst, dus stop niet na de eerste herkenning;

i - 'insensitive': Maak geen onderscheid tussen hoofd- en kleine lettters;

s - 'single': Beschouw de gehele invoer als één regel. Dit betekent dat $(\backslash n)$ geaccepteerd wordt door $(.)$ of $(\star)$;

m - 'multiple': De invoer bestaat uit meer regels. Dit betekent dat $(\hat{})$ en $(\$)$ het begin en einde van elke regel in de tekst aangeven (standaard bij 'sed' en 'grep'). Anders geven zij alleen het begin en einde van de gehele tekst aan;

```
/[a-z]erck/i
```

```
Merck toch hoe sterck
nu in 't werck sich al steld
```

2.5 Uitbreidingen op de reguliere expressies

Sommige programma's bevatten nuttige uitbreidingen en verbeteringen op reguliere expressies, die het herkennen en wijzigen van patronen aanzienlijk gemakkelijker maken. Het is wel mogelijk dat de uitgebreide backtrackmogelijkheden de bovengrens van de verwerkingstijd verhogen van $O(n)$ naar $O(2^n)$.

2.5.1 Niet-gulzige kwantoren

De meeste uitbreidingen voorzien in niet-gulzige kwantoren. Het bereik van deze kwantoren is zo klein mogelijk gemaakt. Een niet-gulzige kwantor is een gulzige kwantor gevolgd door een vraagtekensymbool $(\star?)$, $(+?)$, $(??)$ en $(\{l,h\}?)$.

```
/d.*e/
door al 't mijnen en 't geschut,
dat men daeglijcx hoorde,
menig Spanjaert in sijn hut
in sijn bloed versmoorde.
```

```
/d.*?e/
door al 't mijnen en 't geschut,
dat men daeglijcx hoorde,
menig Spanjaert in sijn hut
in sijn bloed versmoorde.
```

Niet-gulzige kwantoren kunnen weer andere problemen veroorzaken. Bijvoorbeeld "`/[0-9]*?./`" gaat ook op als er geen cijfers zijn. Kijk dus uit voor anorexia met kwantoren.

2.5.2 Bezitterige kwantoren

Bezitterige kwantoren (niet in 'Perl' aanwezig): $(.\star+)$, $(++)$ en $(?+)$ zijn gulzige kwantoren die de gevonden tekst uitsluiten van backtracking. Bijvoorbeeld om bijvoorbeeld de eerste en de laatste "`en`" van "`enenenen`" af te splitsen:

```
s/(en)++(en)/(\1)[\2]/
< enenenen
```

wordt het patoon in zijn geheel geweigerd. Maar met normale gulzige kwantoren en met backtracking is het wel mogelijk:

```
s/(en)+(en)/(\1)[\2]/
< enenenen
> (en)[en]
```

2.5.3 Positief en negatief vooruitkijken

Met geavanceerde realisaties van de NFA zoals 'Perl' en 'Python' is het mogelijk vooruit te kijken naar patronen die nog moeten komen, de 'lookahead'. Een lookahead doet echter niet mee met de verplaatsingen die normaal bij het zoeken en veranderen van teksten plaats vindt. Wel geeft de lookahead informatie over het deel van de tekst volgend op de actuele positie. Er zijn twee manieren van vooruitkijken, met een positieve blik en met een negatieve blik.

Om het eerste voorkomen van een reeks herhaalde patronen te vinden, wordt de positieve lookahead in de constructie "`patroon(?=.*patroon)`" gebruikt:

```
s/ei(?=.*ei)/paasei/
<Een ei is geen ei.
>Een paasei is geen ei.
```

Indien "`ei`" geen deel uitmaakt van een herhaalde reeks, zal de wijziging niet worden uitgevoerd:

```
s/ei(?=.*ei)/paasei/
<Een ei is niet lekker.
>Een ei is niet lekker.
```

Een negatieve lookahead "`(?!patroon)`" is een patroon dat niet op een ander patroon mag volgen. Om bijvoorbeeld het laatste van een reeks herhaalde patronen te vinden, is de constructie "`patroon(?.*`patroon)!" mogelijk:

```
s/ei(?!.*ei)/paasei/
<Een ei is geen ei, twee ei is een half ei, drie ei is een ei.
>Een ei is geen ei, twee ei is een half ei, drie ei is een paasei.
```

2.5.4 Zoeken naar overlappende patronen

Normaal wordt het zoeken hervat na het laatstgevonden patroon. Dit maakt het moeilijk om overlappende patronen te verwerken in bijvoorbeeld DNA-strings:

```
s/([ATCG]{3})/(\1)/g
<ATCGTAGCATCG
>(ATC)(GTA)(GCA)(TCG)
```

Met een positieve lookahead is het wel mogelijk om overlappende patronen te verwerken, bijvoorbeeld de codons van $\{3\}$ letters (behalve de twee laatste letters) in DNA-strings. De stapgrootte van de overlapping wordt met $\{1\}$ aangegeven:

```
s/(?=([ATCG]{3}))[ATCG]{1}/(\1)/g
<ATCGTAGCATCG
>(ATC)(TCG)(CGT)(GTA)(TAG)(AGC)(GCA)(CAT)(ATC)(TCG)CG
```

2.5.5 Positief en negatief achteruitkijken

Het is ook mogelijk terug te kijken naar tekstdelen die al gepasseerd zijn met een positieve- of negatieve ('lookbehind'). Als een beperkt tekstdeel aan een patroon voorafmoet gaan, wordt een positieve lookbehind met "`(?<=tekstdeel)`" gebruikt:

```
s/(?<=half )ei/paasei/
<Een ei is geen ei, twee ei is een half ei, drie ei is een paasei.
>Een ei is geen ei, twee ei is een half paasei, drie ei is een
paasei.
```

Als een beperkt tekstdeel niet aan een patroon vooraf mag gaan dan wordt een negatieve lookbehind met "`(?<!patroon)`" gebruikt:

```
s/(?<!en\s)ei/paasei/g
<Een ei is geen ei, twee ei is een half ei.
```

```
>Een ei is geen ei, twee paasei is een half paasei.
```

Om het backtracken te beperken is het terugkijken alleen met tekstdelen met een vaste lengte mogelijk. Dit betekent dat alternatieven en wildcards wel in de tekstdelen zijn toegestaan, maar kwantoren en verwijzingen niet.

2.5.6 Zoeken naar het n-de patroon

Het "`(?:patroon)`" wordt gebruikt om groepen tussen andere groepen, uit te sluiten van vervanging en verwijzing. Bijvoorbeeld om het n^{de} voorkomen van een patroon te vinden, worden alle voorgaande voorkomens uitgesloten door de constructie "`(((?:.*?patroon){`$n-1$`}.*?)patroon`". Let op, de niet-gulzige kwantorconstructie "`.*?`" is noodzakelijk. Bij het wijzigen wordt het voorafgaande stuk aangegeven met "`\1`":

```
s/((?:.*?ei){3}.*?)ei/\1half ei/
<Een ei is geen ei, twee ei is een ei, drie ei is een paasei.
>Een ei is geen ei, twee ei is een half ei, drie ei is een paasei.
```

2.5.7 Extra mogelijkheden om te wijzigen

Het wijzigen van kleine letters naar hoofdletters met behulp van 'Perl':

```
echo "'t Moedige, bloedige, woedige swaerd" | perl -pe 's/(oe)/\U\1/g'
>'t MOEdige, blOEdige, wOEdige swaerd
```

Het wijzigen van hoofdletters naar kleine letters is in 'Perl' op dezelfde manier mogelijk met "`\L`".

2.5.8 Het specificeren van reguliere expressies

Sommige programma's zoals 'Perl', staan toe dat een reguliere expressie gespecificeerd wordt in meer dan één regel door het argument (x). Daarmee wordt het ook mogelijk om in zo'n specificatie commentaar op te nemen na het (#) symbool:

```
/                  # identificeer URLs in een tekstfile
               \b  # woordgrens
 (https?|ftp|file) # alleen http(s), ftp of file
            :\/\/ # ... gevolgd door ://
```

```
            [^\s]+ # geen spatie, tab of nieuwe regel in de URL
       (?=[\s\.,]) # positieve lookahead naar witte ruimte, punt of comma
/x

Mijn URL is: http://www.mijn_stek.nl/index.html.

Je kan downloaden vanaf ftp://ftp.mijn_stek.nl/pub/.
```

Vergelijk dit eens met:

```
/\b(https?|ftp|file):\/\/[^\s]+(?=[\s\.,])/
```

2.6 Aanbevelingen

Het toepassen van reguliere expressies kan zeer profijtelijk zijn. Vergeet niet dat één reguliere expressie overeen kan komen met meerdere pagina's 'C', 'C++' of 'Java'.

Het nadeel van reguliere expressies is dat de specificatie van één verkeerd karakter of symbool - onopvallend wegens het cryptische en compacte uiterlijk van reguliere expressies - desastreuze gevolgen kan hebben. Bovendien geven de incidentele foutmeldingen tijdens het gebruik te weinig informatie.

Om ontwerpfouten te vinden, is een specificatie met commentaar aan te bevelen. Het blijft echter noodzakelijk om reguliere expressies van te voren op verschillende teksten uit te proberen.

Aanbevolen worden de reguliere expressie bibliotheken op Internet zoals [21]. Echter de documentatie van het aanbod is minimaal. Men blijft verantwoordelijk om zelf de werking te analyseren en te testen.

2.7 Opgaven

1. Wat mankeert er wel- of niet- aan de volgende reguliere expressies?

   (a) "`/$^/`";

   (b) "`/a-Z/`".;

   (c) "`/ha..o/`".;

   (d) "`/kan dit?^$/`";

   (e) "`/(?<!en\s+)/`"

   (f) "`/(?<=e[nm]\s)/`";

(g) Is $(patroon++)$ gelijk aan $(patroon+)+$?

2. DNA woorden bestaan uit de letters *A*,*C*,*G* en *T*. Een aminozuur wordt gecodeerd door een 'codon' dat bestaat uit 3 letters (basenparen).

   (a) Ontwerp een patroon dat begint met het codon "`ATG`" en eindigt met het codon "`GTA`" te vinden. Daartussen mogen alleen codons zitten die voldoen aan:

   - "`CGA`";
   - eindigen op "`CG`";
   - beginnen met "`G`" en eindigen met "`C`" daartussen alleen "`A`" of "`G`".

   (b) Ontwerp een patroon om een telomeer te vinden. Een telomeer is een DNA-deelstring waarin de patronen *TTGGGG* (van micro-organisme's) of *TTAGGG* (van hogere diersoorten), minimaal 2 keer aaneengesloten herhaald wordt. Telomeren zitten aan de uiteinden van een DNA-string (chromosoom). Omdat een telomeer korter wordt bij elke celdeling, geeft de lengte van een telomeer aan of een cel zich nog kan delen. Een cel met een te korte telomeer kan zich niet meer delen en zal sterven zonder nakomelingen.

3. Ontwerp een reguliere expressie om in teksten rationele getallen te vinden. Bijvoorbeeld de getallen: $981 \quad -56 \quad 67, \quad 3,146 \quad -9,99 \quad 0,01 \quad ,123 \quad -,56.$

4. Ontwerp een reguliere expressie voor een emailadres.

5. Wijzig de volgende reguliere expressie zodat ook integers, hexadecimale en octale getallen van verschillende lengte verwisseld kunnen worden.

```
>echo "40 41 42 43" | perl -pe 's/([0-9][0-9]) ([0-9][0-9])/\2 \1/g'
>41 40 43 42
```

6. Ontwerp een reguliere expressie voor een html-link van het type "`<a href="...">...</a>`".

7. Ontwerp een reguliere expressie om van de patronen "`enen`" of "`enenen`" of "`enenenen`" enzovoort, de laatste "`en`" af te splitsen en er een (-) voor te plaatsen. Dus bijvoorbeeld "`enenenen`" wordt "`enenen-en`"

8. Het voorbeeld "`/([a-z]+)([a-z]+) \2([a-z]+)/`" in paragraaf 2.3.3 geeft onverwachte oplossingen.

   (a) Bepaal $(\backslash 2)$ als de laatste letter in het eerste woord het zelfde is als de eerste letter in het tweede woord, zoals in de woorden "`roofkat`" en "`takkebos`";

   (b) Bepaal $(\backslash 2)$ als in de overeenkomstige suffix een herhalend patroon aanwezig is, zoals in de woorden "`grootmama`" en "`mamaatje`".

(c) Verbeter de reguliere expressie zodat wel de grootste gemeenschappelijke deelstring gevonden wordt. Er mogen geavanceerde kwantoren gebruikt worden.

9. Ontwerp een reguliere expressie om in een tekst met meerdere regels de identieke naast elkaar geplaatste woorden te vinden. Bijvoorbeeld: "`In de de zin zijn identieke buurwoorden aanwezig`". Hoofd- en kleine letters mogen geen verschil veroorzaken. Ook regelovergangen mogen geen verschil introduceren. Gebruik de reguliere expressie syntax die gegeven is in appendix B.

10. Maak een reguliere expressie met de om de 2000^{ste} "`TAG`" in een DNA-string te veranderen in "`CAT`". Gebruik de reguliere expressie syntax die gegeven is in appendix B.

Hoofdstuk 3

Formele talen

Kunstmatige talen zoals C/C++, Java, HTML en XML bevatten meer constructies dan reguliere expressies mogelijk maken. Bijvoorbeeld de constructies met haakjes zoals $(\ldots(\ldots))(\ldots)$. Haakjesconstructies zijn in de meeste programmeertalen aanwezig. Zelfs in natuurlijke talen treden verkapte haakjesconstructies op, zoals bijvoorbeeld in het zinsdeel: "*toen (de man (de jankende hond) sloeg)*".

Een ander voorbeeld van de beperktheid van reguliere talen, is de taal $L = \{a^n b^n; 0 < n\}$. Deze relatief eenvoudige woordenschat kan volgens de pompstelling voor reguliere talen niet door een DFA geaccepteerd worden en is volgens stelling 1.1 geen reguliere taal.

3.1 Grammatica's

Een 'grammatica' is niet meer dan een verzameling regels waarmee woorden en zinnen worden gevormd voor een taal. Dat wil niet zeggen dat een taal alleen door één grammatica geproduceerd kan worden. Een taal kan door meer grammatica's geproduceerd worden. Grammatica's die dezelfde taal produceren worden 'equivalent' genoemd.

Definitie 3.1 (Grammatica) *Een grammatica G wordt gedefinieerd als een samenstelling $G = (N, T, P, S)$ waarin:*

N de verzameling symbolen of 'nonterminals' is. Deze symbolen worden voorlopig aangegeven met hoofdletters $A, B, C, \ldots$;

T de verzameling 'terminals', voorlopig aangegeven met kleine letters $a, b, c, \ldots$. Terminals kunnen zowel individuele tekens of rijen van tekens voorstellen. Met de definitie van de terminals en de non-terminals is ook het afgeleide begrip 'vocabulaire' V van een grammatica vastgelegd. Dit is de verzameling terminals en nonterminals van die grammatica: $V = N \cup T$. Als men zinnen beschouwt als rijen van terminals

uit de verzameling $T^\star$, dan zijn 'zinsvormen' - rijen waarin nonterminals en terminals in optreden - elementen van $V^\star$. Willekeurige zinsvormen worden aangegeven met Griekse letters $\alpha, \beta, \gamma \ldots$ behalve de letter ε die gereserveerd is voor een lege rij tekens;

P de eindige verzameling 'productieregels' waarmee nonterminals - eventueel gecombineerd met terminals - overgaan in andere vormen. Deze productieregels worden aangegeven met een $(\rightarrow)$ zoals bijvoorbeeld: $A \rightarrow aA$. Productieregels kunnen beknopt worden weergegeven met de volgende notatie:

$$\alpha \rightarrow (\beta_1, \beta_2, \ldots, \beta_n) \quad i.p.v \quad \alpha \rightarrow \beta_1, \alpha \rightarrow \beta_1, \ldots, \alpha \rightarrow \beta_n$$

S is het 'startsymbool' waarvoor geldt: $S \in V$.

Voorbeeld 3.1 *De productieregels:*

$$N = \{A, B, S\} \quad T = \{a, b\} \quad P = \{S \rightarrow AB, A \rightarrow Aa, B \rightarrow Bb, A \rightarrow a, B \rightarrow b\}$$

worden genoteerd als:

$$N = \{A, B, S\} \quad T = \{a, b\} \quad P = \{S \rightarrow AB, A \rightarrow (Aa, a), B \rightarrow (Bb, b)\}$$

Voorbeeld 3.2 *Een deel van de grammatica van de Engelse taal:*

S	$\rightarrow$	*<sentence>*
<sentence>	$\rightarrow$	*<noun phrase><verb phrase>* |
		<noun phrase><verb phrase><object phrase>
<noun phrase>	$\rightarrow$	*<name>* | *<article><noun>*
<name>	$\rightarrow$	*Jack* | *Jill*
<noun>	$\rightarrow$	*dog* | *horse* | *apple*
<article>	$\rightarrow$	*a* | *an* | *the*
<verb phrase>	$\rightarrow$	*<verb>* | *<adverb><verb>*
<verb>	$\rightarrow$	*eats* | *rides* | *lives* | *beats*
<adverb>	$\rightarrow$	*hardly* | *quickly* | *normally*
<adjective>	$\rightarrow$	*mean* | *slow* | *green* | *old*
<adjective list>	$\rightarrow$	*<adjective><adjective list>* | ε
<object phrase>	$\rightarrow$	*<adjective list><name>* |
		<article><adjective list><noun>

"Jill normally eats a green apple" is volgens deze grammatica correct.

Definitie 3.2 (Taal) *Als uit het startsymbool S door middel van het correct toepassen van een eindig aantal productieregels de terminal a afgeleid kan worden, dan is a een woord van de 'taal' $L(G)$. Alle rijen met tekens die door G geaccepteerd worden vormen de 'woordenschat' $L(G)$.*

$$L(G) = \{a \in T; S \rightarrow \ldots \rightarrow a\}$$

Een nonterminal wordt recursief genoemd als hij zowel in de rechter- en in de linkerkant van een reeks productieregels kan verschijnen. De nonterminal A is:

rechtsrecursief	$A \rightarrow \ldots \rightarrow A\alpha \rightarrow \ldots$
linksrecursief	$A \rightarrow \ldots \rightarrow \alpha A \rightarrow \ldots$
ingebed recursief	$A \rightarrow \ldots \rightarrow \alpha A \beta \rightarrow \ldots$

Recursie leidt tot herhalende patronen, zoals de kwantoren $(\star)$ en $(+)$ bij de reguliere expressies. Het blijkt dat reguliere expressies als equivalente grammatica's geformuleerd kunnen worden:

Voorbeeld 3.3

$r = ab$	$S \rightarrow ab$
$r = a\|b$	$S \rightarrow (a,b)$
$r = a^\star$	$S \rightarrow (\varepsilon, aS)$ *of met* $S \rightarrow (\varepsilon, Sa)$
$r = a^+$	$S \rightarrow (a, aS)$ *of met* $S \rightarrow (a, Sa)$

Voorbeeld 3.4 *De grammatica* $N = \{S\}$, $T = \{a,b\}$, $P = \{S \rightarrow (ab, aSb)\}$ *is equivalent met reguliere expressie* a^+b^+ *die de taal* $L = \{a^n b^n; 0 < n\}$ *produceert. Als men de productieregels* $P = \{S \rightarrow (ab, aSb)\}$ *wijzigt in* $P = \{S \rightarrow (\varepsilon, ab, aSb)\}$, *dan krijgt een equivalente reguliere expressie* $a^\star b^\star$.

Voorbeeld 3.5 *De grammatica* $N = \{S,A\}$, $T = \{a,b\}$, $P = \{S \rightarrow (SaA, aA), A \rightarrow (\varepsilon, bA)\}$ *is equivalent met de reguliere expressie* $(a^+b^\star)^+$.

Omdat recursie meer mogelijkheden biedt dan de kwantoren $(\star)$ en $(+)$, zijn de recursieve productieregels complexer dan reguliere expressies. Om recursie beter te begrijpen, is het verstandig om de grammatica's onder te verdelen in productieregels met een aantal gemeenschappelijke recursieve kenmerken.

3.1.1 Grammaticatypen

Grammatica's zijn door N. Chomsky geclassificeerd naar het type productieregels dat zij gebruiken. Deze hiërarchie is afhankelijk van de mate van vrijheid van de recursie in de productieregels:

type	eigenschap	machine	productieregels
0	recursief aftelbaar	Turingmachine	$\alpha \rightarrow \beta$ geen restricties
1	contextgevoelig	lineair begrensde machine	$\alpha \rightarrow \beta$ $\lvert\alpha\rvert \leq \lvert\beta\rvert$
2	contextvrij	stapelmachine	$A \rightarrow \alpha$
3	regulier	eindige automaat NFA/DFA	$A \rightarrow (a, aA)$ $S \rightarrow \varepsilon$

De eigenschap 'contextvrij' van type 2 geeft aan dat een nonterminal $A \in N$ altijd een zinsvorm α kan produceren, ongeacht de context waarin de nonterminal staat. Contextvrije grammatica's hebben daarom aan de linkerkant van de productieregel één nonterminal en geen andere nonterminals of terminals. Het spreekt vanzelf dat het begrip 'contextgevoelig' gebruikt wordt voor grammatica's met een productieregel waarin de nonterminal A aan de linkerkant niet meer alleen staat.

De contextvrije grammatica's of type 2 grammatica's worden gebruikt voor programmeertalen. Deze productieregels van een type 2 grammatica kunnen redelijk efficiënt gerealiseerd worden met een stapelmachine. Een 'stapelmachine' is een uitbreiding van een eindige automaat met een 'stapel' voor opslag van tijdelijke gegevens. Een stapel is een geheugen met een oneindige capaciteit maar met een beperkte adressering. Geavanceerde realisaties van de stapelmachine zijn de 'bottom-up parser en de 'top-down parser' (par. 3.3 en 3.4).

Voor de Chomsky-hiërarchie geldt: type 3 $\subset$ type 2 $\subset$ type 1 $\subset$ type 0. Dit betekent dat een type 2 wel als type 3 kan werken, maar een type 3 niet als een type 2.

Een contextgevoelige- of type 1 grammatica mag volgens de definitie geen productieregel van het type $\alpha \rightarrow \varepsilon$ bevatten omdat zo'n productieregel niet voldoet aan $|\alpha| \leq |\beta|$. Dit wordt opgelost door het startsymbool S toestemming te geven tot het produceren van een lege rij tekens: $S \rightarrow \varepsilon$.

Voorbeeld 3.6 *Een contextgevoelige grammatica $L(G)$*

$$N = \{S\} \quad T = \{a,b,c\} \quad P = \{S \rightarrow aSb, aS \rightarrow Aa, Aab \rightarrow c\}$$

Na herhaald toepassen van de regel $S \rightarrow aSb$ krijgt men de zinsvorm $a^nSb^n; 0 < n$. Om van die S af te komen, wordt de tweede productieregel toegepast. Daarmee krijgt men het woord: $a^{n-1}Aab^n; 0 < n$. Na de derde regel wordt dit het woord: $a^{n-1}cb^{n-1}; 0 < n$ of anders geschreven: $a^ncb^n; 0 \leq n$.

De recursief aftelbare of type 0 grammatica's zijn alleen met een 'Turingmachine' te realiseren. Een Turingmachine is een automaat, waarbij de overgangsregels niet alleen

leiden tot nieuwe toestanden, maar ook tot het inlezen- en uitvoeren van tekens op een oneindige lange rij geheugencellen, een 'tape' genoemd. Zo'n rij geheugencellen kan volledig geadresseerd worden. In principe kan men elk denkbaar programma met een Turingmachine uitvoeren.

Een 'lineaire begrensde machine' die gebruikt wordt voor het herkennen van een type 1 taal, is een Turingmachine met een adresseerbare rij geheugencellen met beperkte capaciteit. De rij R is lineair begrensd $R \leq c_1 \cdot i + c_2$ en afhankelijk van het totale aantal invoertekens i en de twee constanten c_1 en c_2.

Zowel type 2, 1 en 0 talen hebben onoplosbare problemen die samenhangen met de onbeslisbaarheid van het 'haltprobleem', zoals bijvoorbeeld de onbeslisbaarheid van het 'Post-correspondentieprobleem', genoemd naar E. Post die het voor het eerst formuleerde. De onbeslisbaarheid van het 'Post-correspondentieprobleem' betekent dat er geen systematische methode is die uit twee rijen met geïndexeerde woorden aan kan geven of twee gelijke niet-lege zinnen zijn te vormen in de zelfde volgorde van indexen. Dit probleem stelt directe grenzen aan de taalherkenning met productieregels van het type $\alpha \rightarrow \beta$ en indirect ook grenzen aan automatische correctie van fouten en grenzen aan minder voor de handliggende toepassingen zoals kunstmatige intelligentie, plagiaatcontrole en decompilage.

Voorbeeld 3.7 *De twee woordrijen $v = [abb, bb]$ en $w = [a, bbb]$ over het alfabet $A = \{a, b\}$ hebben bijvoorbeeld de oplossing voor het Post-correspondentieprobleem: $v_1v_2v_2 = abbbbbb = w_1w_2w_2$. Maar voor $v = [ab, aab, aba]$ en $w = [aba, aa, baa]$ is de oplossing een stuk lastiger te vinden, zoniet onmogelijk. Er zijn rijen v en w die sowieso geen oplossing hebben. Dat het 'Post-correspondentieprobleem' onbeslisbaar is, wordt grofweg aangetoond door te bewijzen dat het 'haltprobleem' beslisbaar is als er een methode bestaat dat voor willekeurige tweetallen woordrijen kan beslissen of zij een gemeenschappelijke zinnen hebben of niet [10].*

Een overzicht van problemen waarvan de beslisbaarheid afhankelijk is van het type grammatica, wordt in de volgende tabel gegeven:

omschrijving	probleem	beslisbaar bij type			
		0	1	2	3
woord in taal?	$a \in L(G)$	$\times$	$\times$	$\surd$	$\surd$
taal leeg?	$L(G) = \emptyset$	$\times$	$\times$	$\surd$	$\surd$
taal eindig?	$\lvert L(G)\rvert < n$	$\times$	$\times$	$\surd$	$\surd$
alle mogelijke woorden?	$L(G) = T^\star$	$\times$	$\times$	$\times$	$\surd$
talen equivalent?	$L(G_1) = L(G_2)$	$\times$	$\times$	$\times$	$\surd$
geen overeenkomst?	$L(G_1) \cap L(G_2) = \emptyset$	$\times$	$\times$	$\times$	$\surd$
beperkte overeenkomst?	$\lvert L(G_1) \cap L(G_2)\rvert < n$	$\times$	$\times$	$\times$	$\surd$

De meeste programmeertalen maken gebruik van contextvrije en reguliere talen (type 2 en 3). Dat wil niet zeggen dat contextvrije grammatica's altijd toereikend zijn. Sommige problemen in de programmeertalen zijn alleen oplosbaar met type 0 en 1 grammatica's, zoals ondermeer uit de volgende voorbeelden blijkt:

Voorbeeld 3.8 *Een probleem treedt op bij compilage van broncode als men van een identifier wil weten of de declaratie (de eerste $(a|b)^n$) plaats vindt vóór het gebruik (de tweede $(a|b)^n$). De string $(a|b)^m$ is de daartussen geplaatste broncode. De taal $L = \{(a|b)^n(a|b)^m(a|b)^n; 0 < n, 0 < m\}$ over het alfabet $A = \{a,b\}$ is niet contextvrij.*

Voorbeeld 3.9 *Als de woorden a en c de formele parameterlijst en b en d de actuele parameterlijst van twee procedures voorstellen. Als de formele en actuele parameterlijst evenlang moeten zijn, dan is dit probleem te beschrijven als de taal $a^\star b^\star c^\star d^\star$ met een gelijk aantal a's en c's en een gelijk aantal b's en d's. Echter de taal $L = \{a^n b^m c^n d^m; 0 < n, 0 < m\}$ over het alfabet $A = \{a,b,c,d\}$ is niet contextvrij.*

Dit soort taalproblemen worden bijvoorbeeld in een compiler niet-grammaticaal opgelost met behulp van een 'symbolentabel'. Aan de hand van deze symbolentabel kan gecontroleerd worden of een identifier al gedeclareerd is en of het type van de identifier (een integer of float variabele of procedure) overeenkomt met het gebruik. Met de symbolentabel kan ook gecontroleerd worden of de lengte van actuele parameterlijsten overeenkomt met die van de formele parameterlijsten. Men zou ook gebruik kunnen maken van een contextgevoelige grammatica, maar daarvoor is een opwaardering van de stapelautomaat tot Turingmachine noodzakelijk.

Voorbeeld 3.10 *De contextgevoelige grammatica met de productieregels $S \to (aSBC, aBC)$, $CB \to BC$, $aB \to ab$, $bB \to bb$, $bC \to bc$, $cC \to cc$ genereert het woord $a^2b^2c^2$:*

$$\begin{array}{rcll} S & \to & aSBC & \\ & \to & aaBCBC & (S \to aBC) \\ & \to & aabCBC & (aB \to ab) \\ & \to & aabBCC & (CB \to BC) \\ & \to & aabbCC & (bB \to bb) \\ & \to & aabbcC & (bC \to bc) \\ & \to & aabbcc & (cC \to cc) \end{array}$$

Op soortgelijke manier wordt de taal $a^n b^n c^n; 0 < n$ gegenereerd.

3.2 Contextvrije grammatica's

Omdat de contextgevoelige grammatica's aanleiding kunnen geven tot onvoorspelbaar gedrag en de reguliere expressies te beperkt zijn, beperken programmeertalen zich meestal

tot contextvrije grammatica's waaraan aanvullende eisen zijn gesteld. In een contextvrije grammatica moeten bijvoorbeeld alle nonterminals op een op andere manier afgeleid kunnen worden uit het startssymbool S. En het is noodzakelijk dat elke nonterminal uiteindelijk moeten leiden tot één of meer terminals.

Voorbeeld 3.11 *De contextvrije grammatica* $N = \{S,A,B\}$, $T = \{a,b,c\}$, $P = \{S \rightarrow aA, A \rightarrow (aB), B \rightarrow (bB,a)\}$) *vormt de taal* $L(G_1) = \{a^2b^na; 0 \leq 0\}$. *Alle nonterminals worden gebruikt om woord aaba te vormen:* $S \rightarrow aA \rightarrow a(aB) \rightarrow aa(bB) \rightarrow aaba$. *Hieruit blijkt ook dat alle nonterminals afgeleid kunnen worden uit het startsymbool S.*

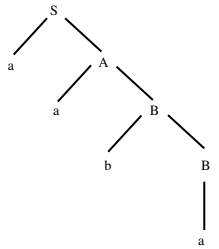
Voorbeeld 3.12 *De contextvrije grammatica* $N = \{S,A,B,C\}$, $T = \{a,b,c\}$, $P = \{S \rightarrow aA, A \rightarrow (aB), B \rightarrow (bB,a), C \rightarrow (c,cB)\}$). *De nonterminal C kan niet afgeleid worden uit het startsymbool S.*

3.2.1 Grammaticale ontleding

Grammatica's worden niet alleen gebruikt om talen te definieren, zij worden ook gebruikt om te controleren of een zin in die taal past. Dit wordt het syntactisch ontleden van een zin genoemd. De stapelmachine die zinnen ontleedt aan de hand van een contextvrije grammatica, wordt met een vakterm 'parser' genoemd.

Tijdens het ontleden van de rij invoertekens aan de hand van een contextvrije grammatica wordt een 'syntaxboom' geproduceerd die de structuur van de rij invoertekens aangeeft. Zo'n syntaxboom heeft in de knopen nonterminals en in de bladeren terminals. De knoop waaraan de rest vastzit wordt wortel genoemd.

Voorbeeld 3.13 *De contextvrije grammatica* $G_1 = (N = \{S,A,B\}, T = \{a,b,c\}, P = \{S \rightarrow aA, A \rightarrow aB, B \rightarrow (bB,a)\})$.



Figuur 3.1: Syntaxboom van G_1

Het woord aaba wordt met de reeks productieregels $S \rightarrow aA \rightarrow aaB \rightarrow aabB \rightarrow aaba$ *herkend als woord van de taal* $L(G_1)$ *en weergegeven als syntaxboom.*

Een complexe syntaxboom betekent dat er veel machinecode wordt gegeneerd door een compiler. De complexiteit van de syntaxbomen kan beperkt worden door het:

verwijderen van ε-productieregels. Elke nonterminal moet leiden tot één of meer terminals. Productieregels die leiden tot een lege rij met tekens $A \rightarrow \ldots \rightarrow \varepsilon$ moeten vervangen worden. Bijvoorbeeld de productieregels $P = \{S \rightarrow (SaA, aA), A \rightarrow (bA, \varepsilon)\}$ worden herschreven tot $P' = \{S \rightarrow (SaA, Sa, aA, a), A \rightarrow (bA, b)\}$;

niet-recursief maken van het startsymbool S. Als het startsymbool S aan de rechterkant van een productieregel staat, moet de regel herschreven worden. Bijvoorbeeld $P = \{S \rightarrow (SaA, Sa, aA, a), A \rightarrow (bA, b)\}$ wordt $P' = \{S \rightarrow B, B \rightarrow (BaA, Ba, aA, a), A \rightarrow (bA, b)\}$;

verwijderen van nutteloze nonterminals. Bijvoorbeeld de productieregels $P = \{A \rightarrow (aA, a, B), B \rightarrow (bB, b)\}$ worden herschreven tot $P' = \{A \rightarrow (aA, a, bB, b),\ B \rightarrow (bB, b)\}$. De originele productieregels P leiden tot langere afleidingen dan die van P'. Bijvoorbeeld $A \rightarrow B \rightarrow b$ is langer dan $A \rightarrow b$.

Definitie 3.3 (Gereduceerde grammatica) *Een grammatica die voldoet aan bovenstaande regels heet een 'gereduceerde grammatica'.*

Voorbeeld 3.14 *Geef de afleiding voor de "aaaa" met* $P = \{S \rightarrow (SaA, aA), A \rightarrow (bA, \varepsilon)\}$ *en met* $P' = \{S \rightarrow B, B \rightarrow (BaA, Ba, aA, a), A \rightarrow (bA, b)\}$. *Voor P:*

$S \rightarrow SaA \rightarrow SaAaA \rightarrow SaAaAaA \rightarrow aAaAaAaA \rightarrow aaAaAaA \rightarrow aaaAaA \rightarrow aaaaA \rightarrow aaaa$

Voor P'*:*

$$S \rightarrow B \rightarrow Ba \rightarrow Baa \rightarrow Baaa \rightarrow aaaa$$

Hoewel de productieregels P' *complexer zijn dan P, geven zij aanleiding tot een kortere afleiding en een minder complexe syntaxboom dan de productieregels P.*

3.2.2 De normaalvormen

Twee grammatica's zijn equivalent als zij dezelfde taal genereren, dus bij dezelfde rij invoertekens identieke syntaxbomen geven. De equivalentie van twee contextvrije grammatica's is in principe onbeslisbaar, er bestaat geen programma of algoritme dat dit automatisch kan bepalen. Daarom moet men dit met andere methoden bepalen. Daarvoor is het handig om grammatica's in een standaard vorm - een 'normaalvorm' - te schrijven. Normaalvormen kunnen gemakkelijk met elkaar vergeleken worden dan grammatica's die niet in een normaalvorm staan. Bovendien gaan analyse en optimaliseringsmethoden van grammatica's uit van normaalvormen. Er zijn twee belangrijke normaalvormen, de 'Chomsky normaalvorm' en de 'Greibach normaalvorm'.

Definitie 3.4 (Chomsky normaalvorm) *Een gereduceerde contextvrije grammatica heeft de Chomsky normaalvorm als elke productieregel één van de volgende vormen heeft:*

1. $A \rightarrow A_1 A_2$
2. $A \rightarrow a$
3. $S \rightarrow \varepsilon$

Het blijkt mogelijk elke gereduceerde contextvrije grammatica in de Chomsky normaalvorm te brengen, door alle productieregels van de vorm $A \rightarrow \gamma \quad 2 \leq |\gamma|$ zoals $A \rightarrow B_1 B_2 \ldots B_n$ te herschrijven met nieuwe nonterminals: $A \rightarrow B_1 D$, $D \rightarrow B_2 E$, $E \rightarrow B_3 F, \ldots$. Productieregels waarvan $|\gamma| < 2$, staan al in de normaalvorm. Productieregels van de vorm $A \rightarrow aB$, worden herschreven als $A \rightarrow XB$ en $X \rightarrow a$ en productieregels zoals $A \rightarrow Bb$, worden herschreven als $A \rightarrow BY$ en $Y \rightarrow b$.

Voorbeeld 3.15 *De productieregels* $P = \{S \rightarrow A, A \rightarrow (aAa, b)\})$ *hebben de volgende Chomsky normaalvorm* $P' = \{S \rightarrow A, A \rightarrow (b, XY), X \rightarrow a, Y \rightarrow AX\})$.

Definitie 3.5 (Greibach normaalvorm) *Een gereduceerde contextvrije grammatica heeft de Greibach normaalvorm als elke productieregel één van de volgende vormen heeft:*

1 $A \rightarrow aA_1 A_2 \ldots A_n$
2 $A \rightarrow a$
3 $S \rightarrow \varepsilon$

Het is mogelijk om elke gereduceerde contextvrije grammatica, die geen lege strings genereert in de Greibach normaalvorm te brengen. De Greibach normaalvorm heeft de eigenschap dat elke productieregel leidt tot één terminal. De lege string ε mag alleen door het startsymbool S geaccepteerd worden. De stapelmachine die de Greibach normaalvorm accepteert is volledig deterministisch. Als alle regels in rij 1 van het type $A \rightarrow aA_1$ zijn, dan is er sprake van een reguliere grammatica.

Voorbeeld 3.16 *De productieregels* $P = \{S \rightarrow (AB), A \rightarrow (aA, bB, b)\}, B \rightarrow b)$ *hebben de volgende Greibach normaalvorm* $P' = \{S' \rightarrow (aAB, bBB, bB), A \rightarrow (aA, bB, b), B \rightarrow b\})$.

Een manier om de Greibach normaalvorm te vinden, is uit te gaan van de Chomsky-normaalvorm. En vervolgens de linksrecursieve productieregels tot rechtsrecursieve productieregels te herschrijven: $A \rightarrow (AB, a)$ tot $A \rightarrow (aB^{\star})$ etc. (zie paragraaf 3.4).

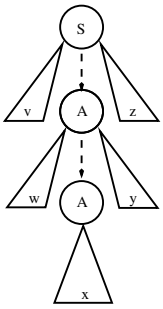
3.2.3 Pompstelling voor contextvrije talen

Om te bepalen of een taal contextvrij is, moet men een stapelmachine construeren die deze taal kan accepteren. Maar om te bewijzen of een taal niet-contextvrij is, moet men aantonen dat er geen stapelmachine is die deze taal kan accepteren. Bij het aantonen dat een taal niet-contextvrij is, gaat men daarom uit van een pompstelling:

Stelling 3.1 (De pompstelling voor contextvrije talen) *Als* $L(G)$ *een contextvrije taal is, dan is er een getal* k *zodanig dat bij elk woord* $k < |u = vwxyz|$ *waarvoor geldt* $|wxy| \leq k$ *en* $0 < |w| + |y|$ *ook de woorden* $vw^n xy^n z; 0 \leq n$ *geaccepteerd worden.*

Het aantal nonterminals in de grammatica van die taal is gelijk aan $|N|$. Als $|u| < |N|$ dan moet er een nonterminal zijn die meer dan één keer gebruikt wordt (het duiventilprincipe)

in de afleidingsreeks $S \rightarrow \ldots \rightarrow u$. Er is dus minstens één nonterminal A waarvoor geldt dat $S \rightarrow \ldots \rightarrow A \rightarrow \ldots \rightarrow A \rightarrow \ldots u$. Deze afleiding kan men voorstellen als $S \rightarrow \alpha \rightarrow (\beta)^+ \rightarrow \gamma \ldots$.



Figuur 3.2: Pompstelling voor contextvrije talen

De productieregels $\alpha \rightarrow (\varepsilon, v)$ en $\gamma \rightarrow (\varepsilon, z)$ staan voor de eventuele lege, niet-herhaalde prefix v en suffix z. De productieregel $\beta \rightarrow (w\beta y, x)$ stelt het herhalende deel voor van de afleiding. Binnen dit herhalende deel staat w voor de prefix, x voor de infix en y voor de suffix. Omdat het herhalende deel geen lege afleidingen kan produceren is het onmogelijk dat w en y beide leeg kunnen zijn. Omdat β onbeperkt herhaald kan worden, is $vw^n xy^n z; 0 \leq n$. En omdat $k < |vw^n xy^n z|$ geldt, moet $|wxy| \leq k$ zijn.

Voorbeeld 3.17 *Volgens de pompstelling geldt voor elk woord $u = a^k b^k c^k = vwxyz$ met $0 < k < |u| = 3k$ en $|wxy| \leq k$ en $0 < |w| + |y|$ dat de taal $L = \{a^n b^n c^n; 0 \leq n\}$ contextvrij is. Als $wy = a^i b^j$ of $wy = b^i c^j$ of $wy = a^i c^j$ met $0 < |w| + |y| = i + j$, dan kan er geen woord $vw^2 xy^2 z$ gemaakt worden waarin het aantal a's, b's en c's gelijk is. Uit deze tegenspraak volgt dat de taal $L = \{a^n b^n c^n; 0 \leq n\}$ niet-contextvrij is.*

3.2.4 Dubbelzinnige contextvrije grammatica's

In een dubbelzinnige grammatica kan een rij invoertekens door twee verschillende reeksen productieregels geaccepteerd worden. Dit veroorzaakt onnodige backtracking, verkeerde syntaxbomen en foutmeldingen. Voor een programmeertaal is dat onacceptabel.

Voorbeeld 3.18 *Gegeven, een grammatica waarbij de nonterminal E een expressie voorstelt, de terminals $+$ en $-$ rekenkundige infix operatoren en a, b getallen of variabelen voorstellen. Deze grammatica met de productieregels $E \rightarrow (E+E, E-E, (E), -E, a, b)$ is voor de $(-)$ operator dubbelzinnig omdat invoertekens $5-2-1$ door de reeks $E \rightarrow E-E \rightarrow (E-E)-E \rightarrow (E-2)-E \rightarrow (5-2)-E \rightarrow (5-2)-1 = 2$ en door de reeks $E \rightarrow E-E \rightarrow E-(E-E) \rightarrow E-(E-E) \rightarrow E-(2-E) \rightarrow 5-(2-1) = 4$ wordt geproduceerd.*

Men kan dubbelzinnigheid van een infix-operator oplossen door bij de grammaticale ontleding aan te geven dat de linkerassociatie $(((a)-b)-c)$ wel is toegestaan en de rechterassociatie $(a-(b-(c)))$ niet is toegestaan. De prioriteit van de linker- of rechterontleding,

met een vakterm ‘operators precedence’ genoemd, heeft het voordeel dat de grammatica niet gewijzigd hoeft te worden. Het opgeven van deze prioriteit is niet bij elke parser mogelijk.

Een grammaticale oplossing voor het dubbelzinnige gedrag van de infix-operator in $E \rightarrow E - E$ is de infix-operator te vervangen door postfix- of prefix-operatoren. Hoewel de resulterende taal door deze ingreep wezenlijk veranderd is, wordt zij wel eenvoudiger omdat er geen haakjes meer nodig zijn. Sommige programmeertalen gebruiken postfix-operatoren zoals ‘Forth’ en ‘Postscript’. Een taal met prefix-operatoren is bijvoorbeeld de taal ‘Lisp’.

Infix:	$E \rightarrow (E+E, E-E, (E), -E, a, b)$	$a-b$, $-a$, $-(a+b)$
Prefix:	$E \rightarrow (+EE, -EE, -E, a, b)$	$-ab$, $-a$, $--+ab$
Postfix:	$E \rightarrow (EE+, EE-, E-, a, b)$	$ab-$, $a-$, $ab+-$

Voorbeeld 3.19 *De grammatica $E \rightarrow (E+T, E*E, i, (E))$ is dubbelzinnig. Want de invoertekens $5*2+1$ worden door de reeks $E \rightarrow E+T \rightarrow (E*E)+T \rightarrow (E*2)+T \rightarrow (5*2)+T \rightarrow (5*2)+1 = 11$ en door de reeks $E \rightarrow E*E \rightarrow 5*E \rightarrow 5*(E) \rightarrow 5*(E+T) \rightarrow 5*(2+T) \rightarrow 5*(2+1) = 15$ geaccepteerd.*

Door de introductie van de nonterminals F (factor) en T (term) kan dubbelzinnigheid ook op andere manieren grammaticaal verholpen worden:

Voorbeeld 3.20 *In plaats van de dubbelzinnige grammatica $E \rightarrow (E+T, E*E, i, (E))$, kan de ondubbelzinnige grammatica $E \rightarrow (E*T, T)$, $T \rightarrow (F+T, F)$, $F \rightarrow (id, (E))$ gebruiken. Ook de grammatica $E \rightarrow (E+T, T)$, $T \rightarrow (T*F, F)$, $F \rightarrow (i, (E))$ is een goede vervanging.*

Voorbeeld 3.21 *De zinsvorm “`if a then if b then c else d`” is dubbelzinnig. Het kan “`if a then (if b then c) else d`” of “`if a then (if b then c else d)`” betekenen.*

Een belangrijke aanwijzing voor dubbelzinnigheid is de aanwezigheid van zowel links- en rechtsrecursieve productieregels: $A \rightarrow A\alpha$ (linksrecursief) en $A \rightarrow \beta A$ (rechtsrecursief) in dezelfde productieregel: $A \rightarrow (A\alpha, \beta A)$. Deze aanwijzing geldt ook voor indirecte afleidingen zoals $A \rightarrow \ldots (A\alpha, \beta A)$. De indirecte afleidingen zijn er voor verantwoordelijk dat het aantonen van dubbelzinnigheid een onbeslisbaar probleem is, verwant met het ‘Post-correspondentieprobleem’.

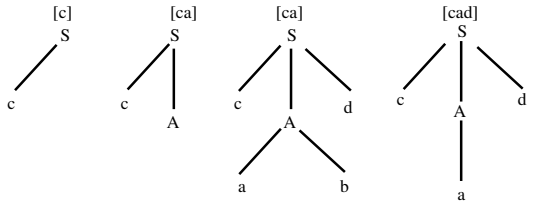
3.3 Top-down parsing

Er bestaan verschillende strategieën bij het ontleden van de syntax. De belangrijkste twee strategieën zijn top-down en de bottom-up ontleding.

De uitdrukking ‘top-down’ wordt verklaard met de constructie van een syntaxboom waarbij eerst bovenin de wortel wordt aangelegd. Vervolgens worden recursief de onderliggende linker- en rechterbomen geplaatst totdat onderaan het laatste blad gecreëerd is.

De top-down parser is een parser die in staat is dubbelzinnigheid te vermijden. De productieregels voor de alternatieve productieregels voor een nonterminal mogen niet leiden tot hetzelfde woord. Dit is noodzakelijk voor een top-down ontleding zonder dubbelzinnigheid. De grammatica die voldoet aan deze eis wordt ‘LL-grammatica’ genoemd.

Voorbeeld 3.22 *De top-down parser met de productieregels $S \rightarrow cAd, A \rightarrow (a, ab)$, krijgt de rij invoertekens “cad”. De parser construeert eerst de wortel S van de syntaxboom. Het eerste invoerteken (c) komt overeen met de terminal van de productieregel $S \rightarrow cAd$. Nu worden de productieregels $A \rightarrow (a, ab)$ mogelijk. Het invoerteken (a) komt overeen met beide varianten van A. Bij inlezen van het derde teken (d), blijkt dat de eerste variant $A \rightarrow a$ gekozen moet worden. Als het derde invoerteken (b) was geweest, zou de tweede variant $A \rightarrow ab$ voldaan hebben. Maar als het derde teken een (a) was geweest, dan had de parser een foutmelding moeten geven.*



Figuur 3.3: Syntaxboom voor *cad*

De keuze voor het correcte alternatief van de A productie is met lookahead informatie mogelijk.

De productieregels die echte problemen veroorzaken bij de top-down ontleding van de grammatica, zijn linksrecursief. De linksrecursieve productieregels veroorzaken onnodig backtracken en geven problemen bij het genereren van de juiste foutmeldingen. Het verwijderen van linksrecursieve productieregels is mogelijk door ze eerst uit te schrijven en vervolgens het (vaak reguliere) patroon opnieuw te formuleren. Linksrecursieve productieregels van de vorm $A \rightarrow (A\alpha, \beta)$ zijn in het algemeen, mits β niet begint met een A, te herleiden tot $A \rightarrow \beta B, B \rightarrow (\alpha B, \varepsilon)$. Eventuele ε-productieregels $B \rightarrow (\alpha B, \varepsilon)$ moeten daarna zo snel mogelijk worden teruggewerkt tot $S \rightarrow \varepsilon, \ldots, B \rightarrow (\alpha B)$.

Voorbeeld 3.23 *De productieregel $A \rightarrow (a, b, Ac)$, wordt uitgeschreven tot:*

stap	*productie*
1	$a\|b\|Ac$
2	$ac\|bc\|Acc$
3	$acc\|bcc\|Accc$
$\vdots$	
n	$ac^n\|bc^n\|Ac^{n+1}$

Dit komt overeen met de reguliere expressie $ac^\star|bc^\star$, die herschreven kan worden als twee rechtsrecursieve productieregels $A \rightarrow (a,b,aB,bB), B \rightarrow (c,cB)$.

Bij top-down syntax ontleding kan men de complexiteit van de syntaxboom verlagen door de productieregels te 'factoriseren'. Factorisatie van productieregels is een eenvoudige herschrijving van $A \rightarrow (\alpha\beta,\alpha\gamma)$ via de reguliere expressie $A = \alpha(\beta|\gamma)$ naar $A \rightarrow (\alpha B), B \rightarrow (\beta,\gamma)$.

Als alle alternatieven van een productieregel $A \rightarrow (A_1,A_2,\ldots A_n)$ met een verschillende 'startterminal' beginnen, dan is grammatica ondubbelzinnig en geschikt voor top-down ontleding. De parser die ondubbelzinnigheid voorkomt door één teken vooruit te kijken en daarmee de gewenste productregel kiest, wordt 'LL(1) parser' genoemd. De uitdrukking 'LL(n)' geeft het aantal invoertekens aan dat gebruikt worden om bij het ontleden vooruit te kijken. Tegenwoordig zijn er LL-parsers die meerdere invoertekens vooruitkijken.

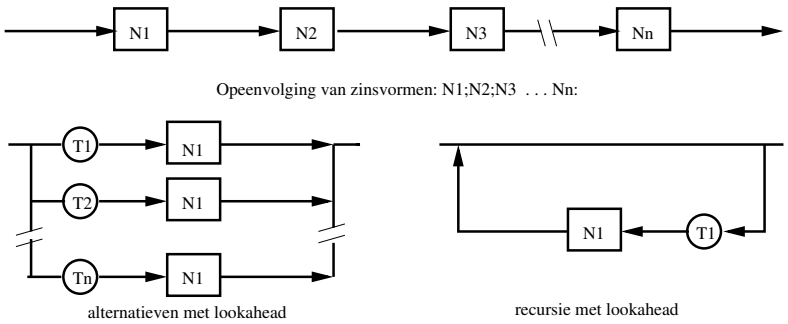
De ondubbelzinnigheid van de LL(1) grammatica kan verklaard worden met het geval $A \rightarrow (bB,cC,\ldots)$. Bij deze productieregel zijn de terminals $b,c,\ldots$ éénduidig gekoppeld aan de alternatieven $B,C,\ldots$ en daarom bepalend voor de keuze van een alternatief. Door deze éénduidigheid is er geen dubbelzinnigheid. De parser hoeft dan ook niet te backtracken, waardoor het van het parsen efficiënter wordt. Als bij productieregels van het type $A \rightarrow (B,C,\ldots)$, de startterminals, 'first-symbolen' genoemd, van $B,C,\ldots$ verschillend zijn, wordt backtracking voorkomen. Voorkomen moet worden dat startterminals in de alternatieven die direct en indirect afgeleid kunnen worden, gelijk aan elkaar zijn. Daarbij zijn vooral de $A \rightarrow \varepsilon$-productieregels belangrijk, omdat die weer kunnen leiden tot nieuwe (indirecte) startterminals, die 'follow-symbolen' worden genoemd.

Voorbeeld 3.24 *De productieregels $E \rightarrow (T,T+E)$, $T \rightarrow (F,(F*T))$, $F \rightarrow ((E),i)$ beschrijven een deel van een eenvoudige rekentaal. In deze productieregels betekent E een 'expressie', T een 'term', F een 'factor' en de terminal i een 'identifier'. De identifier i gerealiseerd door de reguliere expressie $(a|b|c\ldots|z)^+$. De tekens $(+)$, $(*)$, $(($ en $))$ zijn terminals. Deze eenvoudige rekentaal top-down parser is op de volgende wijze gerealiseerd in 'C':*

```
void expressie() { // E -> T, T + E
 term();
 if (invoer[teken]=='+') {teken++; expressie();}
}
```

```
void term() {      // T -> (T * F), F
 if (invoer[teken]==’(’) {
     teken++;
     term();
     if (invoer[teken]==(’*’) {teken++; factor();} else foutmelding();
     if (invoer[teken]==(’)’) teken++; else foutmelding();
 }
 factor();
}
void factor() {    // F -> (E), i
 if (invoer[teken]==’(’) {
     teken++;
     expressie();
     if (invoer[teken]==(’)’) teken++; else foutmelding();
 } else {if(letter(invoer[teken])) teken++; else foutmelding();}
}
```

LL(1) grammatica's zijn grafisch weer te geven, zoals dat is gebeurd bij het ontwerp van de taal 'Pascal' [14]. In figuur 3.4 zijn de grafische 'legoblokken' waarmee een LL(1) grammatica kan worden getekend, aangegeven. De 'legoblokken' in de schema's stellen de zinsvormen voor, de cirkels zijn de startterminals.

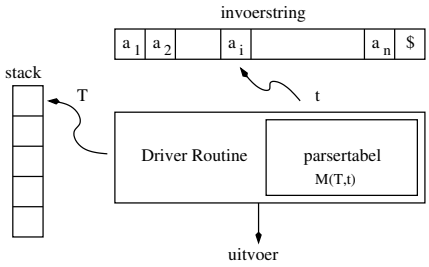


Figuur 3.4: Constructies met lookahead's

Een LL(1)-parser kan ook als een stapelmachine met lookaheadtabel uitgevoerd worden, dit wordt een 'voorspellende parser' genoemd.

Net zoals de top-down parser in broncode, kan de 'voorspellende parser' alleen werken met een LL(1)-taal. Daarnaast moet de voorspellende parser met een extra ($) teken werken. Dit symbool wordt aan het einde van de rij invoertekens en op de lege stapel geplaatst. Het ($) symbool werkt als een afsluiter van een rij tekens en als de bodem van de stapel. Tenslotte is er een tabel M waarmee het ontleden door het symbool τ boven op de stapel en het actuele invoerteken t gestuurd wordt:

1. als $\tau = t =$ \$, dan stopt de parser zonder fouten;



Figuur 3.5: De voorspellende parser

2. als $\tau = t \neq \$$, dan haalt de parser τ van de stapel en leest het volgende invoerteken uit de rij;

3. als τ een nonterminal is, dan wordt $M(\tau,t)$ bepaald. Indien dit de productieregel $M(\tau,t) = \tau \rightarrow \alpha$ is, dan wordt τ op de stapel vervangen door α. Indien $M(\tau,t) = syntaxfout$, dan wordt er een foutmelding gegeven, waarna de parser zich moet herstellen om het ontleden te hervatten.

Het volgende voorbeeld geeft de werking van een voorspellende parser (bron [1]):

Voorbeeld 3.25 *Voor de voorspellende parser moeten eerst de linksrecursieve productieregels in $E \rightarrow (T, T+E)$, $T \rightarrow (F, F*T)$, $F \rightarrow ((E), i)$ vervangen worden. De nieuwe productregels worden: $E \rightarrow TE'$, $E' \rightarrow (\varepsilon, TE')$, $T \rightarrow FT'$, $T' \rightarrow (\varepsilon, *FT')$, $F \rightarrow ((E), i)$.*

Bovendien moeten voor de voorspellende parser eerst alle first- en followsymbolen in een M-tabel worden opgenomen. Om deze verzameling first- en followsymbolen te vinden, worden de volgende regels gebruikt:

$A \rightarrow \ldots \rightarrow a\alpha$ $A \rightarrow (\alpha_1, \alpha_2, \ldots, \alpha_n)$	$first(A) = \{a\}$ $first(A) = first(\alpha_1) \cup first(\alpha_2) \ldots \cup first(\alpha_n)$
regel voor first-symbolen:	$\forall_{1 \leq i < j \leq n} first(\alpha_i) \cap first(\alpha_j) = \emptyset$
$S \rightarrow \ldots \rightarrow \alpha A \beta$	$follow(A) = first(\beta)$
regel voor follow-symbolen:	$A \rightarrow \ldots \rightarrow \varepsilon \quad \Rightarrow \quad first(A) \cap follow(A) = \emptyset$

Bepaal van de nieuwe productieregels $E \rightarrow TE'$, $E' \rightarrow (\varepsilon, TE')$, $T \rightarrow FT'$, $T' \rightarrow (\varepsilon, *FT')$, $F \rightarrow ((E), i)$, de first- en followsymbolen:

$$
\begin{array}{lcl}
first(E \rightarrow TE') & = & first(T \rightarrow FT') = first(F \rightarrow ((E), i)) = \{(, i\} \\
first(E' \rightarrow (\varepsilon, TE')) & = & \{+, \varepsilon\} \\
first(T' \rightarrow (\varepsilon, *FT')) & = & \{*, \varepsilon\} \\
follow(E \rightarrow TE') & = & follow(E' \rightarrow (\varepsilon, TE')) = \{), \$\} \\
follow(T \rightarrow FT') & = & follow(T' \rightarrow (\varepsilon, *FT')) = \{+,), \$\} \\
follow(F \rightarrow ((E), i)) & = & \{+, *,)\$\}
\end{array}
$$

De $M(\tau,t)$ tabel met de first- en followsymbolen is:

L	i	$+$	$*$	$($	$)$	\$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to i$			$F \to (E)$		

Het ontleden van de rij invoertekens "$i+i*i$" geeft de volgende stappen. Stap 1 begint met op de stapel \$E (het startsymbool) te plaatsen en de invoerrij af te sluiten met een \$:

stap	*stapel*	*invoer*	*uitvoer*	*stap*	*stapel*	*invoer*	*uitvoer*
1	$\$E$	$i+i*i\$$		10	$\$E'\ T'\ i$	$i*i\$$	$F \to i$
2	$\$E'\ T$	$i+i*i\$$	$E \to TE'$	11	$\$E'\ T'$	$*i\$$	
3	$\$E'\ T'\ F$	$i+i*i\$$	$T \to FT'$	12	$\$E\ T'\ F'\ *$	$*i\$$	$T \to *FT'$
4	$\$E'\ T'\ i$	$i+i*i\$$	$F \to i$	13	$\$E\ T'\ F$	$i\$$	
5	$\$E'\ T'$	$+i*i\$$		14	$\$E'\ T'\ i$	$i\$$	$F \to i$
6	$\$E'$	$+i*i\$$	$T' \to \varepsilon$	15	$\$E'\ T'$	\$	
7	$\$E'\ T\ +$	$+i*i\$$	$E' \to +TE'$	16	$\$E'$	\$	$T' \to \varepsilon$
8	$\$E'\ T$	$i*i\$$		17	\$	\$	$E' \to \varepsilon$
9	$\$E'\ T'\ F$	$i*i\$$	$T \to FT'$		\$		

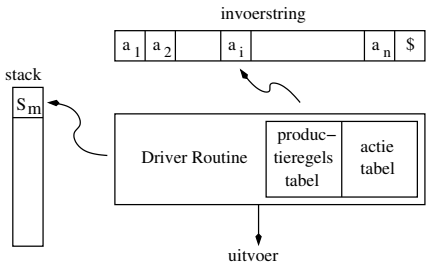
In stap 1 is de voorspellende parser in de begintoestand \$E, met i als eerste invoerteken. De nonterminal E wordt van de stapel gehaald en vervangen door de rechterkant van de productieregel in de tabel $M(\tau = E,t = i) = E \to TE'$. Dit geeft op de stapel $E'T$ (in omgekeerde volgorde \$$E'T$). In stap 2 wordt de nonterminal T van de stapel gehaald en vervangen door de rechterkant van $M(\tau = T,t = i) = T \to FT'$ (in omgekeerde volgorde \$$E'T'F$). In stap 3 wordt de nonterminal F vervangen door $M(\tau = T,t = i) = F \to i$, het volgende teken t wordt uit de invoerrij gehaald. Dit herhaalt zich tot de stapel alleen het symbool \$ bevat.

Indien de rij invoertekens bij een top-down parser tot een syntaxfout leidt, zoals bijvoorbeeld het ontbreken van een identifier *i* na een $(+)$, zal de top-down parser een vervangend teken van het zelfde type op de stapel plaatsen en een foutmelding geven. De top-down ontleding kan dan weer hervat worden met de rest van de rij invoertekens. Bij syntaxfouten die op deze manier niet hersteld kunnen worden, kan een synchronisatie-teken $(;)$ of een einde-regel-teken $(\backslash n)$ gebruikt worden om het ontleden weer te hervatten.

3.4 Bottom-up parsing

Bottom-up parsers maken gebruik van het 'schuif-reduceer principe'. Dit houdt in dat de invoertekens op de stapel geplaatst worden, het 'schuiven', totdat voldaan is aan de rechterkant van een van productieregel. Zodra dit optreedt, worden de betreffende tekens van

de stapel gehaald en vervangen door de nonterminal aan de linkerkant van de betreffende productieregel, dit heet 'reduceren'. Uiteindelijk moet de stapel bij de laatste nonterminal leeg gemaakt zijn. Het effect van reduceren en schuiven wordt 'links-rechts ontleding' genoemd. De stackmachine die dit uitvoert is de 'LR-parser'.



Figuur 3.6: De LR-parser

Bij elke reductie op de stapel wordt een knoop van de syntaxboom gecreëerd die de nonterminal voorstelt van de rechterkant van de productieregel. De kind-knopen zijn op zich weer deelbomen met gereduceerde teminals en nonterminals. De terminals zijn dan zijn het de bladeren en de nonterminals zijn dan de knopen. De term 'bottom-up' is de volgorde van de opbouw van de syntaxboom, die onderaan begint bij de bladeren en uiteindelijk bovenaan stopt bij de creatie van de wortel.

Als er productieregels zijn waarvan de prefixen van de linkerkanten overeen komen, dan treden er problemen op. Indien zo'n prefix boven op de stapel ligt, dan moet er gekozen worden tussen een reductie *(neem (non)terminals van de stapel, reduceer (non)terminals, leg het reductieresultaat op de stapel)*) of een schuifactie (*leg het invoerteken op de stapel)*). De productieregels zijn dan niet beslissend meer. Moet er bijvoorbeeld gereduceerd worden als bij de productieregels $A \rightarrow (BB, BBa, BBC)$ de prefix BB boven op de stapel ligt, of moet er eerst het volgende teken ingelezen worden? De oplossing ligt in het gebruik van lookaheads.

Het gedrag van de LR-parser is afhankelijk van de stapel en het invoerteken en twee tabellen: een 'actie'- en een 'goto'-tabel. In de actie-tabel staan de volgende elementen:

element	omschrijving
sn	de volgende toestand is *sn*
rn	reduceer de stapel met regel *rn*
acc	accepteer de string

In de goto-tabel staat de volgende toestand van de LR-parser.

Het volgende voorbeeld (bron [1]) geeft de werking van een eenvoudige LR(0)-parser zonder lookahead's:

Voorbeeld 3.26 *De grammatica van een eenvoudige rekentaal heeft de volgende productieregels, genummerd van* 1 *t/m* 6*:*

reduceer	*productieregel*	*reduceer*	*productieregel*
$r1$	$E \rightarrow E + T$	$r4$	$T \rightarrow F$
$r2$	$E \rightarrow T$	$r5$	$F \rightarrow (E)$
$r3$	$T \rightarrow F * T$	$r6$	$F \rightarrow i$

De actie-tabel en de goto-tabel zijn gecombineerd tot:

toestand	*actie*						*goto*		
S	i	$+$	$*$	$($	$)$	\$	E	T	F
0	$s5$			$s4$			1	2	3
1		$s6$				*acc*			
2		$r2$	$s7$		$r2$	$r2$			
3		$r4$	$r4$		$r4$	$r4$			
4	$s5$			$s4$			8	2	3
5		$r6$	$r6$		$r6$	$r6$			
6	$s5$			$s4$				9	3
7	$s5$			$s4$					10
8		$s6$			$s11$				
9		$r1$	$s7$		$r1$	$r1$			
10		$r3$	$r3$		$r3$	$r3$			
11		$r5$	$r5$		$r5$	$r5$			

Het ontleden van de syntax vindt plaats op de stapel en wordt gestuurd door de tabellen:

stap	*stapel*	*invoer*	*stap*	*stapel*	*invoer*
1	0	$i*i+i\$$	8	0 T 2	$+i\$$
2	0 i 5	$*i+i\$$	9	0 E 1	$+i\$$
3	0 F 3	$*i+i\$$	10	0 E 1 $+$ 6	$i\$$
4	0 T 2	$*i+i\$$	11	0 E 1 $+$ 6 i 5	\$
5	0 T 2 $*$ 7	$i+i\$$	12	0 E 1 $+$ 6 F 3	\$
6	0 T 2 $*$ 7 i 5	$+i\$$	13	0 E 1 $+$ 6 T 9	\$
7	0 T 2 $*$ 7 F 10	$+i\$$	14	0 E 1	\$

In stap 1 *is de parser in de begintoestand* 0*, met het i als eerste invoerteken. In de actietabel geeft rij* 0 *en kolom i, de waarde s*5*, dat betekent schuif invoer en plaats op de stapel de toestandswaarde* 5*. In stap* 2 *is de stapel* $[0\ i\ 5]$ *en* (i) *is verdwenen uit de invoerrij. Vervolgens is* $(*)$ *het volgende teken en de actie van toestand* 5 *is de reductie* $F \rightarrow i$*. De tekens* (5) *en* (i) *worden van de stapel gehaald waardoor toestand* 0 *weer zichtbaar wordt. Omdat de goto-tabel voor toestand* 0 *en F de toestand* 3 *geeft, worden F en* 3 *op de stapel geplaatst. De parser staat nu in stap* 3*. De rest van de schuif-reduceer activiteiten verlopen op dezelfde manier tot dat het acc element in stap* 14 *gevonden wordt.*

Indien er in de invoerstring syntaxfouten zijn, zoals bijvoorbeeld het ontbreken van een identifier i na een $(+)$ *teken, dan zal de parser in de actietabel de schuif- of reduceeropdracht niet kunnen vinden. Er moet dan een foutactie ondernomen worden, die een*

foutmelding genereert en het ontleden weer herstelt. De lege posities in de actie-tabel worden gebruikt voor de foutacties e_x.

toestand	*actie*						*goto*		
S	i	$+$	$*$	$($	$)$	\$	E	T	F
0	$s5$	$e1$	$e1$	$s4$	$e2$	$e42$	1	2	3
1	$e11$	$s6$	$e31$	$e40$	$e19$	acc			
$\vdots$									

De LR(0)-parser is beperkt tot eenvoudige productieregels. Daarentegen kan een volledige LR(n)-parser in principe meer grammatica's ontleden dan een top-down parser. Een nadeel van een volledige LR(n)-parser is de zeer grote tabel. Daarom wordt meestal gewerkt met een bijzondere LR(1)-parser, de 'lookahead-linksrechtsparser' ('LALR(1)') parser met operators-precedence. Bij het gebruik van één lookahead en operators-precedence is het mogelijk is het aantal toestanden (de rijen in de actie/goto-tabellen) te combineren tot één toestand (rij). Om de tabellen voor een LALR(1) parser te genereren uit productieregels, maakt men gebruik van 'parsergenerators' zoals 'yacc', , 'bison' en'CUP'.

3.5 Opgaven

1. Waarom maakt men meestal geen gebruik van type 0 en 1 grammatica's bij programmeertalen?

2. Is een reguliere taal contextvrij? Verklaar uw antwoord.

3. Geef van de grammatica $N = \{S,A\}$, $T = \{a,b\}$, $P = \{S \rightarrow (a,aAS), A \rightarrow bS\}$) de syntaxboom voor het woord *abaabaa*.

4. (a) Ontwerp een contextvrije grammatica voor palindromen met een even aantal tekens voor het alfabet $A = \{a,b\}$. Palindromen zijn woorden waarin de tekens van voor-naar-achteren dezelfde volgorde hebben als van achteren-naar-voren. Voorbeelden zijn 'anna', 'otto' 'parterretrap' en 12344321;

   (b) Ontwerp een contextvrije grammatica voor palindromen maar nu voor een oneven aantal tekens. Voorbeelden zijn 'ada', 'lepel', 'racecar' en 1234321;

   (c) Ontwerp een contextvrije grammatica voor even- en oneven palindromen.

5. Van welk type is de productie $Axy \rightarrow axbyc$. Geef een toepassing. Verklaar uw antwoord.

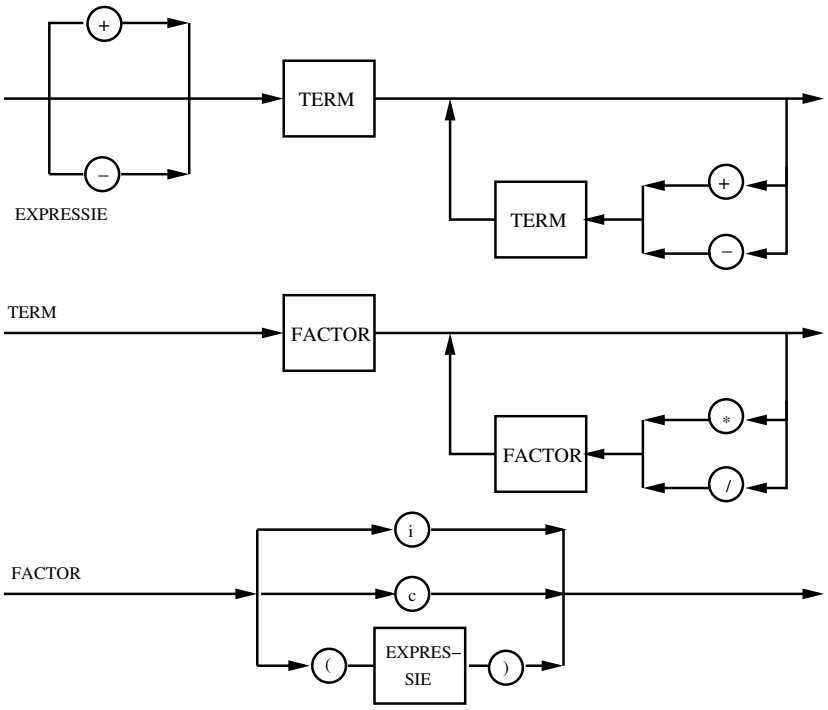
   .

6. De taal $L = \{a^n b^n c^n; 0 < n\}$ over het alfabet $A = \{a,b,c\}$ is niet contextvrij.

(a) Verklaar dit;

(b) Geef een afleiding voor het woord $a^3b^3c^3$ met de grammatica in voorbeeld 3.10.

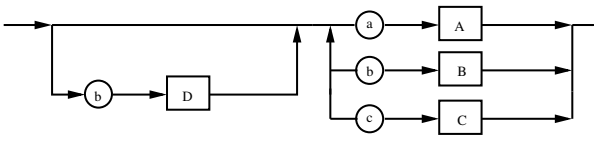
7. Geef de productieregels die passen bij figuur 3.7.



Figuur 3.7: Schema 1 met productieregels

8. Analyseer figuur 3.8 en geef commentaar.

.



Figuur 3.8: Schema 2 met productieregels

Hoofdstuk 4

Vertalers

Een vertaler die een brontaal in een doeltaal vertaalt, wordt een ‘compiler’ genoemd als de brontaal een programmeertaal is. De doeltaal van een compiler bestaat uit een rij uitvoertekens, die opdrachten voor de doelmachine voorstellen. Een rij opdrachten of instructies wordt een ‘programma’ genoemd.

De doeltaal kan een taal zijn waarvan de instructies rechtstreeks door de machine gelezen en uitgevoerd kan worden (‘machinetaal’ of ‘objectcode’) of het kan een doeltaal zijn voor een andere machine. Een compiler voor een andere machine wordt een ‘cross-compiler’ genoemd.

Het is ook mogelijk dat een compiler een doeltaal vertaalt voor een programma, dat zich gedraagt als een machine en daarom een ‘virtuele machine’ wordt genoemd. De taal voor zo’n virtuele machine wordt ‘tussentaal’ genoemd en de instructies worden ‘pseudo-instructies’ genoemd.

Indien het vertalen van broncode naar tussencode gecombineerd wordt met het direct uitvoeren van de tussentaal door zo’n virtuele machine, dan is er sprake van een ‘just-in-time-compiler’, ookwel ‘on-the-fly-compiler’ genoemd. Deze compilers worden vaak gebruikt bij scriptingtalen zoals ‘Perl’[17], ‘Python’[18],‘PHP’[19] en ‘TCL’[20].

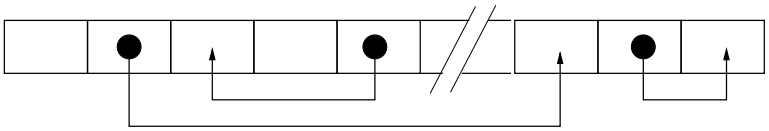
In tegenstelling tot een ‘just-in-time’ compiler wordt bij een ‘interpreter’ geen doelcode gegenereerd. Een interpreter vertaalt de programmeertaal niet in een doeltaal, maar voert de instructies tijdens het vertaalproces gelijk uit. Het voordeel van een interpreter is, dat informatie die tijdens het uitvoeren van de instructies beschikbaar komt, weer gebruikt kan worden bij de vertaling van de rest van het programma.

Voorbeeld 4.1 *Interpreters kunnen tijdens het uitvoeren bepalen of een variabele gebruikt wordt als integer of als float. Dit is een van de redenen waarom interpreters minder zware eisen stellen aan de type-declaratie van variabelen (‘weak typing’).*

Interpreters kunnen zeer efficiënt inspelen op de informatie die tijdens het uitvoeren van de instructies vrij komt, iets wat compilers niet kunnen. Het nadeel van een interpreter is,

dat het uitvoeren van de instructies regelmatig onderbroken moet worden voor vertaalactiviteiten.

Compilers kunnen daarentegen, de lengte van de rij uitvoertekens beperken en optimaliseren omdat zij tijdens het vertalen informatie hebben die verderop aanwezig is in de rij invoertekens. Daarnaast kunnen compilers programmeerfouten vinden, door variabelen te controleren op de declaratie en het gebruik van het type ('strong typing'). Gecompileerde programma's worden bovendien sneller uitgevoerd dan geïnterpreteerde programma's. Met andere woorden, een compiler heeft informatie over de gehele rij invoertekens en een interpreter heeft informatie over de reeds uitvoerde instructies.



Figuur 4.1: Verwijzingen

In de meeste machinetalen zijn spronginstructies en verwijzingen naar datastructuren aanwezig. Deze informatie is pas bekend na de vertaalslag waarin de totale lengte en de indeling van de uitvoerrij vast is komen te liggen. Pas daarna kan de vertaalslag komen waarin deze informatie verwerkt wordt in de verwijzingen. Bij sommige compilers wordt de laatste vertaalslag door een 'assembler' uitgevoerd. De assembler vertaalt - leesbare machine-instructies - naar binaire machinetaal. Assemblertaal bevat symbolische verwijzingen, ookwel 'labels' genoemd, die door de assembler worden vervangen door indexen van de posities in de uitvoerrij.

De 'just-in-time-compilers' zitten eigenlijk tussen de compiler en de interpreter in. Zij combineren de voordelen van het compileren met de directe uitvoering van een interpeter, zonder veel te verliezen aan uitvoeringssnelheid.

4.1 De BNF-notatie voor practische vertalers

Practische vertalers maken gebruik van reguliere expressies en van contextvrije grammatica's. De wiskundige notatie die bij reguliere expressies en grammatica's wordt gebruikt, is alleen handig als men de achtergronden van de grammatica wil bestuderen. Door het beperken van hoofdletters voor nonterminals, is men bij een realistische toepassing al gauw door de voorraad heen. Net zoals bij patroonherkennende programmatuur met reguliere expressies, wordt voor de contextvrije grammatica een andere, meer praktische, notatie gebruikt. Dit is meestal de 'Backus Naur notatie' ('BNF'). Deze BNF-notatie wordt gedefinieerd met de volgende omzettingstabel:

terminals	$a-z$	*letterlijk*
nonterminals	$A-Z$	$<$ *nonterminal* $>$
productieregels	$\rightarrow$	$::=$ of $->$ of :
escaping	$<$ $>$ $\mid$ $::=$	$'<'$ $'>'$ $'\mid'$ $'::='$
alternatieven	$(\alpha,\beta,\ldots)$	$\alpha\mid\beta\mid\ldots$

Voorbeeld 4.2
```
<digit>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<number> ::= <digit> | <digit> <number>
```

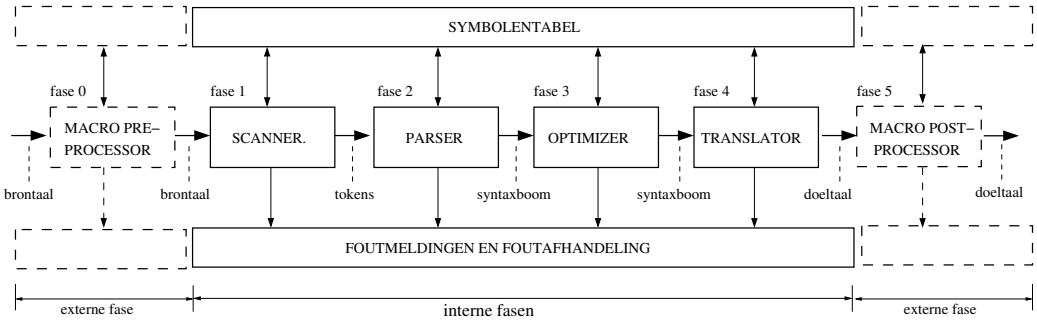
Er zijn ook andere versies van de 'BNF', zoals de 'Extended BNF'. In EBNF worden alle letterlijke teksten tussen ($'$) tekens geplaatst en de nonterminals worden zonder ($<$) en ($>$) tekens aangegeven. Bovendien zijn de kwantoren $\star$, $+$ en ? mogelijk. Commentaar wordt tussen haakjes geplaatst.

Voorbeeld 4.3
```
description    ::= '<rdf:Description' idAboutAttr? propAttr* '/>'
                 | '<rdf:Description' idAboutAttr? propAttr* '>'
                       propertyElt* '</rdf:Description>'
                 | typedNode
propertyElt    ::= '<' propName '>' value '</' propName '>'
                 | '<' propName resourceAttr? propAttr* '/>'
propAttr       ::= propName '="' string '"'
                     (with embedded quotes escaped)
typedNode      ::= '<' typeName idAboutAttr? propAttr* '/>'
                 | '<' typeName idAboutAttr? propAttr* '>'
                       property* '</' typeName '>'
```

'EBNF' wordt vooral gebruikt bij de formele definities van protocollen, dataformaten en opmaaktalen zoals 'XML' en 'SGML'.

4.2 Vertaalfasen

Het vertaalproces wordt verdeeld in fasen. In elke fase worden deeltaken uitgevoerd. De volgorde van deze fasen is onveranderlijk. Wel verschillen vertalers in het aantal keren dat zij de fasering opnieuw aflopen voordat de brontaal vertaald is in de doeltaal. De fasen zijn verdeeld in interne- en externe fasen. Een interne fase wordt door de compiler doorlopen. Een externe fase kan in principe zelfstandig door andere programma's uitgevoerd worden. Vertaalopdrachten voeren deze externe vertalingen integraal uit indien dat nodig is.



Figuur 4.2: De fasen van een vertaling

fase	naam	plaats	functie
0	macro-preprocessor	extern	macrovertaling van brontaal naar brontaal
1	scanner	intern	lexicografische vertaling van brontaal naar tokens
2	parser	intern	grammaticale vertaling van tokens naar syntaxboom
3	optimalisator	intern	vertaling van syntaxboom naar syntaxboom
4	codegenerator	intern	vertaling syntaxboom naar doeltaal
5	macro-postprocessor	extern	macrovertaling van doeltaal naar doeltaal

Het aantal herhaald passeren van de interne fasen door de compiler, verschilt van één keer voor een ‘single-pass compiler‘ tot meerdere keren bij een ‘multi-pass compiler’. Een vertaalslag wordt herhaald als de benodigde informatie pas na de vorige slag bekend is.

Voorbeeld 4.4 *De afmeting van de rij uitvoertekens in de doeltaal is pas bekend nadat de brontaal minimaal één keer vertaald is. Indien in de rij uitvoertekens allerlei verwijzingen zijn naar de lengte van deze rij, kunnen die pas in de volgende vertaalslag worden ingevuld. Deze verwijzingen zijn bijvoorbeeld adressen (indexen in de uitvoerrij) van variabelen, methods, classes en arrays,*

De meeste compilers werken met een contextvrije parser. Dit betekent dat contextgevoelige problemen zoals declaraties van variabelen en het gebruiken van templates met behulp van een ‘symbolentabel’ of met macro’s moeten worden opgelost. De symbolentabel en de afhandeling van fouten en foutmeldingen is verbonden met alle fasen in het vertaalproces. Macro’s worden verwerkt door een ‘preprocessor’ in de externe fase 0 en door een ‘postprocessor’ in de externe fase 5. Deze postprocessor is bij een compiler meestal een ‘macro-assembler’.

4.3 Macrovertaling

Een macrovertaler vervangt een rij invoertekens door een rij uitvoertekens in dezelfde taal. De rij uitvoertekens kan groter zijn dan de rij invoertekens.

Een ‘macro’ of een ‘template’ wordt gedefinieerd met een naam, nul of meer formele parameters en een inhoud. Deelrijen in de invoerrij, die overeenkomen met de naam van een macro, worden vervangen door de inhoud van deze macro. Hierbij worden ook de formele parameters vervangen door actuele parameters. Zo’n contextgevoelige vervanging van formele parameters door actuele parameters is in principe alleen mogelijk met productieregels van het type $\alpha \rightarrow \beta$ waarbij $|\alpha| \leq \beta|$. Hoewel een macrovertaler beperkter is dan een contextgevoelige vertaler, is zij zeer efficiënt en geeft zij een programmeertaal extra mogelijkheden. De belangrijkste mogelijkheden van macro’s zijn:

definitie	aanroep	uitvoer
$A \equiv \varepsilon$	A	
	aAp	ap
$A \equiv a$	A	a
	aAp	aap
$A(\#_1,\#_2) \equiv a\#_1 b\#_2$	$A(1,2)$	$a1b2$
$A(\#_1) \equiv a\#_1 A(\#_1)$	$A(1)$	$a1A(1) = a1a1A(1) = a1a1a1A(1) = \ldots$
	$ifd(A,1,2)$	1 als A gedefinieerd is
	$ifd(A,1,2)$	2 als A ongedefinieerd is
$A \equiv a, B \equiv b$	$if(A,B,C)$	c als $a = b$
$C \equiv c, D \equiv d$	$if(A,B,C,D)$	d als $a \neq b$

In realistische macrovertalers zijn reguliere- en numerieke kwantoren, rekenkundige uitdrukkingen en het invoegen van andere invoerrijen mogelijk.

Voorbeeld 4.5 *De macro “*`define(tenzij(#1),if(!(#1)))`*” wordt bij de aanroep “*`tenzij(10<x)`*” geëxpandeerd tot “*`if(!(10<x))`*”*

Een zelfstandige macro-preprocessor is ‘m4’ [16]. De macro-preprocessor ‘cpp’ is geïntegreerd in [33]. Een geïntegreerde macro-postprocessor is de macro-assembler ‘gas’ die opgenomen is in ‘gcc’ [34]. Een zelfstandige macro-assembler voor de IA-32 architectuur is ‘nasm’ [35].

4.4 Lexicografische vertaling

Bij het parsen van grammatica’s worden teminals gebruikt die eigenlijk grotere eenheden voorstellen. Zoals de terminal *i* die gebruikt wordt voor “`<identifier>`”. Waarom is deze *i* niet volledig met zijn productieregels opgenomen in de grammatica? Omdat terminals die met reguliere expressies te definieren zijn, efficiënter door een NFA/DFA herkend kunnen worden dan door een contextvrije grammatica.

In de eerste fase van de vertaling, de ‘lexicografische vertaling’, worden deze terminals gemaakt uit de invoerrij brontekens door een ‘scanner’. Een ‘scannergenerator’ genereert

uit een verzameling reguliere expressies een scanner in ‘C’ of ‘C++’-broncode. Deze broncode heet “`lex.yy.c`”, en bestaat uit de functie “`int yylex()`” en DFA tabellen voor patroonherkenning.

Deze functie “`int yylex()`” wordt aangeroepen door de ‘parser’ in de volgende fase en geeft bij elke aanroep een token af. Deze tokens zijn regulier vertaalde tekens uit de brontaal naar het alfabet van de parser. Tekens in de invoerrij die niet herkend worden door een reguliere expressie, worden rechtstreeks aan de parser doorgegeven.

Voorbeeld 4.6

```
%{           /* C of C++ declaraties van variabelen en functies */
#include "FILE_NAME.tab.h" /* tokendefinities uit de FILE_NAME.y parser */
%}            /* hier beginnen de hulpdefinities in flex-stijl */
cijfer        [0-9]
getal         {cijfer}+
commentaar    ";"[^\n]*    /* niet-gulzig versie commentaar */
witruimte     [ \t]+
%%
{getal}       {yylval = atoi(yytext); return NUM ;}
"+"           {return PLUS;}
"-"           {return MIN;}
"/"           {return DIV;}
"*"           {return MUL;}
{commentaar}  /* eet commentaar op */
{witruimte}   /* eet witte ruimte op */
.             { error("illegaal symbool", yytext[0]); }
%%            /*$ hier beginnen de extra voorzieningen */
```

Voor het genereren van scanners in de talen ‘C’ en ‘C++’, zijn de scannergenerators ‘lex’ [24] en de moderne GNU versie met de naam ‘flex’ [25] geschikt. Voor het genereren van een scanner in de taal ‘Java’, kan men het programma ‘Jlex’ [26] gebruiken.

4.5 Syntax ontleding

De ‘parser’ is het centrale gedeelte van de vertaler. Zij roept de scanner aan met de functie “`int yylex()`” om het volgende token te parsen. Uiteindelijk vertaalt zij deze tokens naar syntaxbomen of geeft zij foutmeldingen. Een parser is ingewikkelder dan een scanner. Daarom maakt een ‘parsergenerator’ van een verzameling productieregels met de naam “`FILE_NAME.y`” een LALR(1) parser in C met de naam “`'FILE_NAME.tab.c`”. Deze parser bestaat uit de functie “`int yyparse()`” die werkt met LALR(1) actie-goto-tabellen en een tokendefinities die in de naam “`'FILE_NAME.tab.h`” staan.

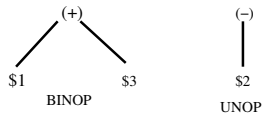
Voorbeeld 4.7

```
%{ /* declaraties van yylex(), yyeror() en syntaxbomen */
```

```
#include "syntaxtree.h"
%} /* declaraties van tokens en nonterminals in bison-stijl */
%union  {int num; string id; Exp exp;} /* YYSTYPE (stapeltype) = C union */
%token <id>  ID      /* type id token identifier */
%token <num> NUM     /* type num token NUM  */
%type  <exp> e       /* type exp non-terminal e */
%start e             /* start non-terminal e */
%left  PLUS MIN      /* linksassociatief token: a+-b+-c = (((a)+-b)+-c) */
%left  MUL DIV       /* linksassociatief token: a*/b*/c = (((a)*/b)*/c) */
%right UMIN          /* rechtsassociatief token: -a = -(a) */
%%
e : e PLUS e         {$$=binop(A_Plus,$1,$3);}
  | e MINUS e        {$$=binop(A_Minus,$1,$3);}
  | e MUL e          {$$=binop(A_Times,$1,$3);}
  | e DIV e          {$$=binop(A_Div,$1,$3);}
  | NUM              {$$=numExp($1);}
  | ID               {$$=idExp($1);}
  | MIN e %prec UMIN {$$=unOp(A_UnMin,$2);}  /* declaratie token UMUN */
  | '(' e ')'        {$$=$2;}
  ;                  /* einde grammatica $ */
%%
```

De productieregels staan in een aangepaste BNF-notatie, gevolgd door een actie-declaratie tussen $\{$ en $\}$. In de actie-declaratie kan men naar actuele alternatief van een productieregel verwijzen met \$\$ voor het linkerdeel en met 1, 2, 3, ... naar het rechterdeel. Bijvoorbeeld, in de actie-declaraties voor de alternatieven van de productieregel $A \rightarrow (a, aB, aBc)$ zou men kunnen verwijzen met \$\$ $\rightarrow$ \$1, \$1\$2, \$1\$2\$3.

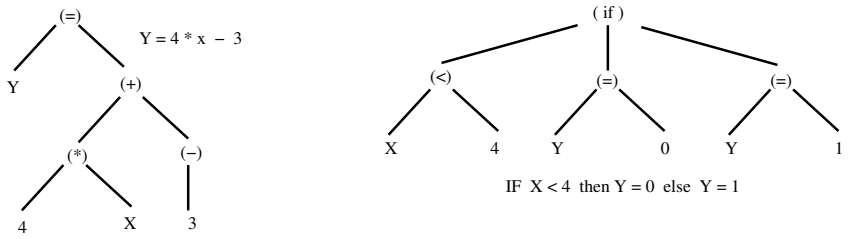
Een actie wordt 'geactiveerd' met de uitdrukking "`$$=actie`". Als de parser onderdeel is van een interpreter, dan worden in zo'n actie de betreffende instructies uitgevoerd. Als de parser deel uitmaakt van een compiler dan is de actie het 'knopen' van deelbomen aan de syntaxboom. De constructie "`$$=binop($2,$1,$3)`" bij de productieregel "`E: E + E`" geeft aan, dat de deelboom met de knoop $(+)$ en de kinderen $(\$1)$ en $(\$3)$ aan de syntaxboom wordt geknoopt.



Figuur 4.3: Knopen in de syntaxboom

Voor het genereren van parsers in de talen 'C' en 'C++', zijn de parsergenerators 'yacc' [27] en de moderne versie'bison' [28] geschikt. Voor het genereren van een parser in de taal 'Java', kan men het programma 'CUP' [29] gebruiken.

Voorbeeld 4.8



Figuur 4.4: Voorbeeld syntaxbomen

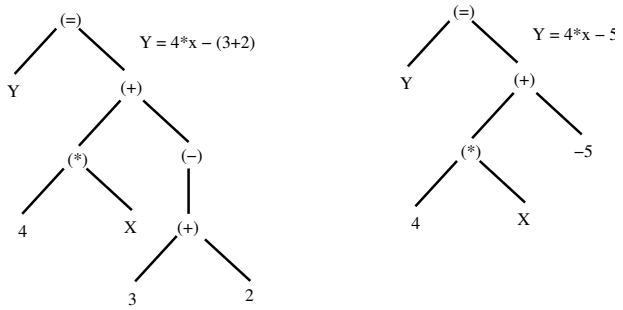
4.6 Optimalisatie

De syntaxbomen geven de grammaticale structuur van de rij invoertokens. Met deze syntaxbomen kan men de rij uitvoertekens in assembler- of machinetaal genereren. Maar voordat dit gebeurt, moet de syntaxboom worden geoptimaliseerd. Optimalisatie kan tactisch of strategisch zijn.

4.6.1 Strategische optimalisatie

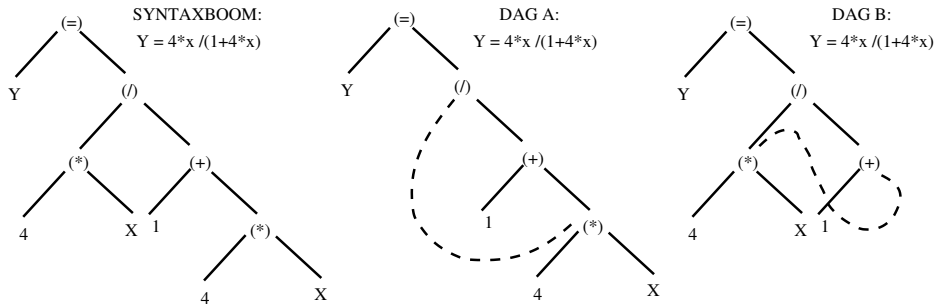
De ‘strategische optimalisatie’ in fase 3 transformeert de syntaxboom voordat er code gegenereerd wordt. Zij is veel doelmatiger dan de tactische optimalisatie die wordt behandeld in paragraaf 4.6.2. Omdat syntaxbomen meestal in het werkgeheugen worden opgebouwd, zijn ze gemakkelijk te wijzigen. In - en aan - syntaxbomen kan geknipt en geplakt worden. Enkele voorbeelden van strategische optimalisatie met knippen en plakken zijn:

- Het vervangen van deelexpressies door constanten. Bijvoorbeeld $y = 4 * x - (3 + 2)$ wordt $y = 4 * x - 5$:



Figuur 4.5: Vervangen van een deelboom door een blad

- Het omvormen van een syntaxboom omgevormd tot een ‘directed acyclic graph’, een ‘DAG’, om gemeenschappelijke delen van de expressie $y = 4 * x/(1 + 4 * x)$ opnieuw te gebruiken.



Figuur 4.6: Gemeenschappelijke delen

  Hoewel de DAG geen boomstructuur meer heeft, kan zij op dezelfde manier bewandeld worden als een normale syntaxboom. Alleen wordt een (gestippelde) tak niet bewandeld maar vervangen door een verwijzing naar het gemeenschappelijke deel.

- Het beperken van het aantal niveaus in een syntaxboom. Deze optimalisatie probeert de syntaxboom of de DAG zo symmetrisch mogelijk te maken in verband met het aantal registers (zie par. 4.7.2). In figuur 4.6 zou DAG B de voorkeur hebben;

- Het combineren van gemeenschappelijk toekenningen is lijkt op het gebruik van gemeenschappelijke delen. Deze ‘peephole-optimalisatie’ combineert de toekenningen $X = E$; en $Y = E$; tot de toekenning $Y = X = E$;.

Er zijn nog meer strategische optimalisaties mogelijk, zoals het verwijderen van invariante delen uit de for- en while-loop’s en het uitrollen van loops voor het verbeteren van de snelheidprestatie.

4.6.2 Tactische optimalisatie

De ‘tactische optimalisatie’ wordt uitgevoerd tijdens fase 4, de fase waarin de doeltaal gegenereerd wordt. Maar om het verschil met strategische optimalisatie aan te geven, wordt zij nu behandeld. De tactische optimalisatie werkt niet met syntaxbomen, maar met de instructies op het niveau van de machinetaal, zoals bijvoorbeeld het vervangen van sterke- door zwakke-operatoren bij integer-expressies.

$x = x+1$	$x++$	incrementeren, mits deze instructie bestaat
$x = x-1$	$x--$	decrementeren, mits deze instructie bestaat
$x = 0$	$x \oplus x$	'*xor*' met x zichzelf
x^n	$x*x*\ldots*x$	als $0 < n <$ *constante*
$x*2^n$	$x << n$	n keer een 0-bit rechts inschuiven
$x/2^n$	$x >> n$	n keer een bit links uitschuiven
$x\%2^n$	$x\&(2^n-1)$	'*mod*' vervangen door '*and*' instructie

4.7 Machinetaal instructies

Het belangrijkste van de 'translator' in fase 4 is het genereren van instructies in machinetaal. Twee belangrijke typen instructies zijn de stapel- en de registergeoriënteerde instructies. Om het vertaalproces te verklaren, wordt in de voorbeelden een macro-assembleertaal opgebouwd in een gelaagde functionaliteit zodat de algemene principes duidelijk worden. Deze voorbeeld macro-assembleertaal is voor mensen leesbaar en accepteert macro's.

4.7.1 Een stapelgeoriënteerde machinetaal

Om een indruk te krijgen van een machinetaal, wordt een eenvoudige realisatie van de Turingmachine met een stapel gebruikt. De machine heeft een rekenkundige en logische eenheid ('ALU'), een instructiewijzer I, een geheugen met adresseerbare cellen ookwel 'heap' genoemd, een stapel, een vlaggenregister F (de nul-vlag z en de carry-vlag c) en één of meer in- en uitvoerregisters die als geheugencellen geadresseerd kunnen worden. Indien er met positieve en negatieve getallen ('integers') gewerkt wordt, dan moet het vlaggenregister F uitgebreid worden met de sign-vlag s en de overflow-vlag o. In appendix E wordt de relatie tussen de vlaggen c, s, z, o en de numerieke bewerkingen behandeld.

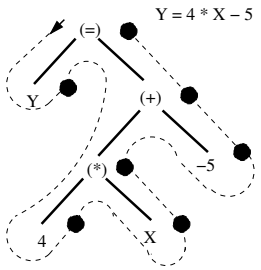
Het geheugen in deze eenvoudige stapelmachine wordt gebruikt voor de opslag van het programma, de stapel en de variabelen. Directe adressering is eenvoudig: een geheugencel i heeft het adres i in de inhoud (i). De waarde (i) wordt aan een naam gekoppeld met een macro-achtige constructie: $X \equiv (i)$, Het adres van de variabele $\#X \equiv i$.

De machine begint met de instructiewijzer I op de eerste instructie en een lege stapel. Daarna wordt het programma, de rij instructies, één voor één uitgevoerd. Om te beginnen zijn de volgende instructies mogelijk:

instructie	stapel-voor	stapel-na	omschrijving
add	$[\ldots,e1,e2]$	$[\ldots,e1+e2]$	optelling op de stapel
sub	$[\ldots,e1,e2]$	$[\ldots,e1-e2]$	aftrekking op de stapel
mul	$[\ldots,e1,e2]$	$[\ldots,e1*e2]$	vermenigvuldiging op de stapel
div	$[\ldots,e1,e2]$	$[\ldots,e1/e2]$	deling op de stapel
and	$[\ldots,e1,e2]$	$[\ldots,e1\&e2]$	'and' op de stapel
or	$[\ldots,e1,e2]$	$[\ldots,e1\|e2]$	'or' op de stapel
xor	$[\ldots,e1,e2]$	$[\ldots,e1\oplus e2]$	'xor' op de stapel
mod	$[\ldots,e1,e2]$	$[\ldots,e1\%e2]$	rest van deling op de stapel
shl	$[\ldots,e1,e2]$	$[\ldots,e1<<e2]$	linksschuiven op de stapel
shr	$[\ldots,e1,e2]$	$[\ldots,e1>>e2]$	rechtsschuiven op de stapel
sto	$[\ldots,e1,e2]$	$[\ldots]$	van stapel naar geheugen, $(e1)\leftarrow e2$
push C	$[\ldots]$	$[\ldots,C]$	constante of celadres op de stapel
push (C)	$[\ldots]$	$[\ldots,(C)]$	inhoud van cel op de stapel
pop (C)	$[\ldots,e]$	$[\ldots]$	neem e van de stapel in cel, $(C)\leftarrow e$
cmp	$[\ldots,e1,e2]$	$[\ldots]$	neem $e2$ en $e1$ van de stapel, $F\leftarrow status(e1-e2)$
jmp C_l	$[\ldots]$	$[\ldots]$	instructiewijzer wordt label, $I\leftarrow C_l$
je C_l	$[\ldots]$	$[\ldots]$	$I\leftarrow C_l$ als $z=1$ dan $(e1-e2)=0$ (zie app. E)
jb C_l	$[\ldots]$	$[\ldots]$	$I\leftarrow C_l$ als $c=1$ dan $(e1-e2)<0$ (absoluut)
ja C_l	$[\ldots]$	$[\ldots]$	$I\leftarrow C_l$ als $c\vee z=0$ dan $(e1-e2)>0$ (absoluut)
jl C_l	$[\ldots]$	$[\ldots]$	$I\leftarrow C_l$ als $o\neq s$ dan $(e1-e2)<0$ (integer)
jg C_l	$[\ldots]$	$[\ldots]$	$I\leftarrow C_l$ als $o=s\wedge z=0$ dan $(e1-e2)>0$ (integer)
call C_l	$[\ldots]$	$[\ldots,C_r]$	inhoud C_r instructiewijzer I op de stapel, $I\leftarrow C_l$
ret	$[\ldots,C_r]$	$[\ldots]$	neem label C_r van de stapel, $I\leftarrow C_r$

Het vertalen van de syntaxbomen is te verklaren als het in post-orde wandelen door de syntaxboom. Tijdens deze wandelingen genereert de translator op bepaalde punten machinetaalinstructies.

Voorbeeld 4.9 *Een rekenkundige expressie wordt vertaalt naar stapelgeoriënteerde machinetaal:*

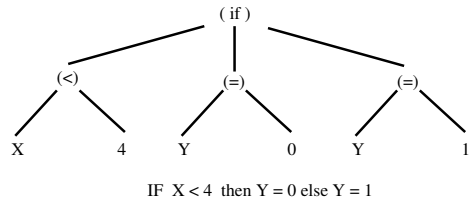


Figuur 4.7: Eenvoudige rekenkundige expressie

stap	*instructie*	*werking*	*stapel*	*X*	*Y*
1	*push #Y*	*leg het adres van geheugencel Y op de stapel*	$[10]$	3	
2	*push* 4	*leg het getal* 4 *op de stapel*	$[10,4]$	3	
3	*push X*	*leg de inhoud van geheugencel X op de stapel*	$[10,4,3]$	3	
4	*mul*	*neem twee getallen van de stapel, vermenigvulig ze en leg het resultaat op de stapel*	$[10,12]$	3	
5	*push* -5	*leg het getal* -5 *op de stapel*	$[10,12,-5]$	3	
6	*add*	*neem twee getallen van de stapel, tel ze op en leg het resultaat op de stapel*	$[10,7]$	3	
7	*sto*	*neem twee getallen van de stapel, plaats het eerste getal in de geheugencel met het adres dat gelijk is aan het tweede getal*	$[\,]$	3	7
8	*end*	`end EQU sub I,1`			
9	*var* $=3$	`X EQU (9) #Y EQU 9`			
10	*var* $=?$	`Y EQU (10) #Y EQU 10`			
11	*vrij geheugen*				

Opmerking: *Assemblermacro's van het type* $X \equiv (9)$ *worden in de programma's aangegeven met "*`X EQU (9)`*".*

Voorbeeld 4.10 *Een conditionele expressie wordt vertaald naar een stapelgeoriënteerde machinetaal:*



Figuur 4.8: Een conditionele uitdrukking

stap	*instructie*	*werking*	*stapel*
1	*push X*	*leg de inhoud van geheugencel* $X = (13) = 3$ *op de stapel*	[3]
2	*push* 4	*leg het getal* 4 *op de stapel*	[3,4]
3	*cmp*	*neem twee getallen van de stapel, trek ze van elkaar af bepaal de bepaal de status F*	[ ]
4	*jl* 11	*als* $(X-4) < 0$ *naar* 10 *(app. E)*	
5	*push #Y*	*leg het adres van geheugencel Y op de stapel*	[14]
6	*push* 0	*leg het getal* 0 *op de stapel*	[14,0]
7	*sto*	*neem twee getallen van de stapel, plaats het eerste getal in de geheugencel met het adres dat gelijk is aan het tweede getal*	[ ]
8	*jmp* 12	*ga naar* 12	
9	*push #Y*	*leg het adres van geheugencel Y op de stapel*	[14]
10	*push* 1	*leg het getal* 1 *op de stapel*	[14,1]
11	*sto*	*neem twee getallen van de stapel, plaats het eerste getal in de geheugencel met het adres dat gelijk is aan het tweede getal*	[ ]
12	*end*	`end EQU jmp 12`	
13	*var* $= 3$	`X EQU (13) #X EQU 13`	
14	*var* $=?$	`Y EQU (14) #Y EQU 14`	
15	*vrij geheugen*		

Virtuele machines worden meestal als stapelgeoriënteerde machines gerealiseerd. Bijvoorbeeld, de 'Java Bytecodes' van Sun en de Common Immediate Language ('CIL') van MicroSoft [31]. Daarentegen zijn talen die rechtstreeks door machines verwerkt worden, vaak registergeoriënteerd.

Virtuele machines worden belangrijker, de hardware is snel genoeg om de lagere prestaties die vroeger aan virtuele machines werd toegeschreven, te compenseren. Virtuele machines zijn vooral voordelig voor het ontwerp, het hergebruik en het onderhoud van de programmatuur. De 'Java Virtual Machine', bijvoorbeeld is een virtuele machine die geschreven is voor één taal, 'Java' en voor meerdere machines. Alle programmatuur geschreven in 'Java' kan op diverse machines worden uitgevoerd.

Een ander voorbeeld is de Common Language Infrastructure ('CLI'), een uitgebreide virtuele machine voor meerdere talen (o.a. 'C'/'C++'/'C#', 'Java#', 'Visual Basic', 'Eiffel', 'Perl'). De 'CLI' is welliswaar beperkt tot de 'IA-32/64' machine-architectuur, maar het is een uitgebreide virtuele machine met extra ondersteuning voor file-formaten en snelle toegang tot onderliggende hardware.

4.7.2 Een registergeoriënteerde machinetaal

Naast de instructieteller I en het vlaggenregister F heeft een registermachine de beschikking over extra registers. Dit betekent dat de instructieverzameling van de stapelgeoriënteerde machine wordt uitgebreid met extra mogelijkheden. Men zou zo'n registermachine kunnen realiseren met bijzondere geheugencellen waarin rechtstreeks rekenkundige en logische bewerkingen plaats kunnen vinden:

instructie	werking
$mov\ (i), j$	$(i) \leftarrow j$
$mov\ (i), (j)$	$(i) \leftarrow (j)$
$add\ (i), j$	$(i) \leftarrow (i) + j$
$add\ (i), (j)$	$(i) \leftarrow (i) + (j)$
$sub\ (i), j$	$(i) \leftarrow (i - j)$
$\vdots$	$\vdots$
$cmp\ (i), j$	$F \leftarrow status((i) - j)$
$cmp\ (i), (j)$	$F \leftarrow status((i) - (j))$
$tst\ i$	$F \leftarrow status(i - 0)$
$tst\ (i)$	$F \leftarrow status((i) - 0)$
$\vdots$	$\vdots$
$jmp\ i$	$I \leftarrow i$
$jmp\ (i)$	$I \leftarrow (i)$
$call\ i$	$I \downarrow, I \leftarrow i$
$call\ (i)$	$I \downarrow, I \leftarrow (i)$
ret	$\uparrow I$

Om in de voorbeelden de doeltaal en de machine eenvoudig te houden, zou men de registers kunnen realiseren met macro's: $R_1 \equiv (i)$, $R_2 \equiv (i+1)$, $R_3 \equiv (i+2)$ etc. Dus als $R_1 \equiv (200)$ dan betekent de instructie $movR_1, 15$, dat de inhoud van geheugencel 200, bekend als R_1, de waarde 15 krijgt. In de 'IA-32'-architectuur hebben registercellen extra rekenkundige en logische mogelijkheden, die normale geheugencellen niet hebben. Om die reden zijn ze in de hardware naast de centrale rekenkundige en logische eenheid geplaatst en hebben ze eigen namen.

Voorbeeld 4.11 *De rekenkundige expressie in figuur 4.7 wordt vertaalt naar een registergeoriënteerde machinetaal:*

stap	*instructie*	*werking*
1	*mov* $R_1, 4$	$R_1 = 4$
2	*mul* R_1, X	$R_1 = R_1 * X$
3	*mov* $R_2, -5$	$R_2 = -5$
4	*add* R_2, R_1	$R_2 = R_2 + R_1 = -5 + 4 * X$
5	*mov* Y, R_2	$Y = 4 * X - 5$
6	*end*	
7	*reg*	*R1 EQU (7)*
8	*reg*	*R2 EQU (8)*
9	$var = 3$	*X EQU (9)*
10	$var =?$	*Y EQU (10)*
11	*vrij geheugen*	

Naast de rekenkundige en logische mogelijkheden hebben registers $R_1 \ldots R_n$ extra mogelijkheden om te adresseren zoals indirecte adressering: $(R) \equiv ((i))$, waarvoor de algebraïsche regel geldt: $(\#R) = R$. Een register kan dan als een wijzervariabele (een 'pointer') gebruikt worden.

Voorbeeld 4.12 *Het adres $\#R \equiv i$ van de registervariabele $R \equiv (i)$ is een wijzer naar een geheugencel i. De constructie $(R) \equiv ((i))$ betekent dat de inhoud geheugencel i, dus ook van het register R, een adres is naar een andere geheugencel:*

stap	*instructie*	*werking*
1	*mov* $R, \#Y$	$R = \#Y$
2	*mov* $(R), 0$	$(R) = (\#Y) = Y = 0$

Met indirecte adressering kan men gemakkelijk rijen van geheugen cellen adresseren. Daarmee worden ook array's mogelijk.

De meeste machines zijn registermachines. Compilers die een brontaal in meerdere doeltalen kunnen compileren zoals 'gcc' [33], maakt in de laatste vertaalslag gebruik van de Register Transfer Language (RTL) [36]. Een verzameling macro's om snel een nieuwe machinetaal te adopteren.

4.8 Procedures, argumenten en interrupts

Bij de register-machine was er sprake van een instructiewijzer en een stapel. Om de principes van procedures en argumentenoverdracht te verklaren, wordt de stapelwijzer ('stackpointer') geïntroduceerd. Om argumenten over te dragen bij functies en procedures heeft de stapelwijzer bijzondere adresseringsmogelijkheden nodig.[1] Niet moet de stapelwijzer

[1] In de 'IA-32'-architectuur heeft het stapelregister geen uitgebreide adresseringsmogelijkheden. Daarom wordt eerst het stapelregister *ESP* naar het register *EBP* gekopieerd die dit wel heeft [37].

indirecte adressering aan kunnen zoals (S), maar ook indexering zoals $(S+4)$ en pre- en post adressering zoals $(--S)$ of $(S++)$.

De stapel wordt gerealiseerd in het vrije geheugendeel, door de stapelwijzer S het bodemadres van de stapel te geven. De rest van het vrije geheugen, de 'heap', dat eventueel opgevraagd en vrijgegeven kan worden, is voor variabelen en andere datastructuren zoals strings en array's. De 'heap' is dus niet hetzelfde als de abstracte datastructuur, die bekend is onder de naam 'heap'.

Bij het plaatsen van een element op de stapel, wordt eerst de stapelwijzer $--S$ verlaagd , vervolgens wordt de inhoud van geheugencel (C) op de stapel $(--S)$ gelegd met de instructie "`push`". Deze pre-adresserings instructie is een macro "`push #1`" met de inhoud "`mov (--s),#1`".

Bij het nemen van een element van de stapel, wordt eerst het element van de stapel in geheugencel C geplaatst, vervolgens wordt de stapelwijzer met 1 verhoogd $S++$, door de instructie "`pop`". Deze post-adresserings instructie is een macro "`pop #1`" met de inhoud "`mov #1,(s++)`".

Door bij een *push*-instructie de stapelwijzer vooraf te verlagen en bij een *pop*-instructie de stapelwijzer achteraf te verhogen, heeft de stapelbodem een hoger adres dan de rest van de stapel. Deze manier is gebruikelijk bij de meeste processoren. Er is echter niets op tegen om dit andersom te doen, mits de stapelwijzer (S) altijd naar de eerste vrije plaats op de stapel wijst.

De *call*-instructie is voor te stellen als twee aaneengesloten (stap wordt niet verhoogd) deelinstructies:

stap	instructie	werking
2206	*call label*	$mov\ (--s), 2207\ =\ push\ 2207$
2206		$mov\ I, label\ =\ jmp\ label$
2207	...	

De *ret*-instructie is voor te stellen als een $pop\ I$ instructie:

stap	instructie	werking
3116	*ret*	$mov\ I, (s++)\ =\ pop\ I$

4.8.1 Aanroepen van functies met argumenten

Functies zonder neveneffecten worden vaak bij recursieve programma's gebruikt. Functies geven meestal een terugkeerwaarde (return value) meestal via de stapel of via een vast register. De overdacht van de waarde van de argumenten wordt 'call by value' genoemd. In de meeste talen worden deze argumentwaarden via de stapel aan de functie overgedragen.

```
;-------------------------------------------------------------------------
start     ...
          push  0         ; terugkeer waarde element (reservering)
          push  label     ; terugkeer adres  element
          push  Y         ; leg de waarde van argument 2 op de stapel
          push  X         ; leg de waarde van argument 1 op de stapel
          push  4         ; aantal argumenten + 2
          call  functie   ; aanroep functie, terugkeer adres op stapel
label     pop   Z         ; Z=functie(X,Y)
          ...
          end             ;
;-------------------------------------------------------------------------
                          ;             stapel = [  r,  i, a2, a1,  4,  i]
                          ;             index       6,  5,  4,  3,  2,  1
functie   mov  r,a1       ; r = a1      r  EQU (S+6)  a1 EQU (S+3)
          add  r,a2       ; r = r + a2  a2 EQU (S+4)
          ...             ;             stapel = [  r,  i, a2, a1,  4,  i]
          add  S,(S+2)    ; S = S + 4   stapel = [  r,  i]
          ret             ; terugkeerwaarde  r = a1 + a2
;-------------------------------------------------------------------------
```

4.8.2 Aanroepen van procedures met argumenten

Functies met neveneffecten zijn geen echte functies, daarom worden zij procedures of subroutines genoemd. Een procedure met een uitvoervariabele als argument, wordt aangeroepen met het adres van de uitvoervariabele op de stapel. De overdacht van het adres van een argument wordt ‘call by reference’ genoemd. In de meeste talen wordt zo’n adres via de stapel aan de procedure overgedragen.

```
;-------------------------------------------------------------------------
start     ...
          push  label     ; terugkeer adres element
          push  X         ; leg de waarde van argument 2 op de stapel
          push  #Y        ; leg het adres van argument 1 op de stapel
          push  4         ; aantal argumenten + 2
          call  procedure ; aanroep procedure
label     ...             ; procedure(Y,X):  Y=Y+X
          ...
          end             ;
;-------------------------------------------------------------------------
                          ;             stapel = [i, a2, #a1,  4,  i]
                          ;             index     5,  4,  3,   2,  1
```

```
procedure mov  a1,1       ; a1 =(#a1) = ((S+3)) = 1   a1 EQU ((S+3))
          add  a1,a2      ; a1 = a1 + a2              a2 EQU (S+4)
          ...             ;             stapel = [i, a2, #a1,  4,  i]
          add  S,(S+2)    ; S = S + 4   stapel = [i]
          ret             ; neveneffect a1 = a1 + a2
;-------------------------------------------------------------------------
```

4.8.3 Interrupts, threading en parallele processen

Een interrupt is een soort *call*-instructie die op een willekeurig moment kan worden aangeroepen door een gebeurtenis buiten het programma. Het initiatief hiervoor ligt meestal bij de in- en uitvoerapparatuur zoals netwerkkaarten, toetsenborden, printers en tijdmelders.

De *int*-instructie komt overeen met een "`push F`" gevolgd door een "`call int_procedure`". De *push*-instructie is noodzakelijk om de statusvlaggen van het geïnterupteerde programma te beschermen die in de "`int_procedure`" worden aangetast. Omdat bij de afsluiting van de "`int_procedure`" het terugkeeradres en de statusvlaggen weer van de stapel gehaald moeten worden, is een *rti*-instructie noodzakelijk. De *ret*-instructie zoals bij een normale procedure of functie gebruikelijk is, voldoet niet omdat het alleen het terugkeeradres zonder de statusvlaggen van de stapel neemt.

Vaak kan men via de in- en uitvoerregisters aangeven of men bepaalde interrupts wil toelaten of blokkeren. Om tijdens een kritiek gedeelte van het programma alle interrupts te blokkeren, moet in het vlaggenregister F een interruptvlag i de waarde 0 krijgen.

Een spontane software-interrupt kan optreden door een fout in het programma zoals een overflow van een integervariabele (app. E), een deling door nul, een illegale instructie of een adressering buiten het toegestane geheugen- of stapelgebied. Daarnaast is het mogelijk om een geprogrammeerde software-interrupt instructie uit te voeren.

Vaak worden geprogrammeerde software-interrupts gebruikt voor het aanroepen van beschermde systeemprogramma's (in kernel-mode) door onbeschermde applicatieprogramma's (in user-mode) en omgekeerd. In dat geval wordt bij de software-interrupt naar een vast adres gesprongen (een interruptvector) waar een andere registerverzameling wordt ingeschakeld. Waardoor het lijkt alsof het interruptprogramma waarnaar toegesprongen wordt, door een andere processor wordt uitgevoerd.

Moderne machines hebben de beschikking over meerdere processoren waarmee het mogelijk is om gelijktijdig meerdere delen van hetzelfde programma uit te voeren. Eigenlijk is er al sprake van pseudo parallellisme bij de interrupts. Het begrip 'threading' is een abstractie van parallelle en pseudo-parallelle processen. Een thread is een programmadeel of beter gezegd een proces dat (eventueel pseudo) gelijktijdig met andere programmadelen uitgevoerd kan worden. Omdat threading geen gebruik maakt van het kostbare multitasking van het operating system is het efficiënter dan multitasking. Voor threading moeten in de machinetaal extra voorzieningen aanwezig zijn van het type:

start T	start een thread
stop T	stop een thread
suspend T	laat een thread wachten
resume T	hervat een thread
destroy T	vernietig een thread

Om te voorkomen dat meerdere processen gelijktijdig dezelfde datastructuren wijzigen, zijn er semafoor-instructies nodig, vergelijkbaar met het blokkeren en accepteren van interrupts (locking).

4.9 Datastructuren

Tot nu toe zijn alleen instructies behandeld die op variabelen werken of op de loop van het programma. Er zijn ook bijzondere constructies met variabelen mogelijk, de 'datastructuren'. Voor datastructuren zijn extra voorzieningen in de symbolentabel nodig. Bovendien moeten er declaratie-opdrachten zijn die datastructuren in het geheugen kunnen reserveren met startwaarden.

Het 'laadprogramma' vult voordat het programma begint, de geheugencellen in het 'code-segment' met instructies en de geheugencellen in het 'data-segment' met de variabelen en hun startwaarden. Tenslotte plaatst het laadprogramma de instructiewijzer op de eerste instructie en kan het programma beginnen.

4.9.1 De symbolentabel

Tijdens de lexicografische analyse in fase 0 worden identifiers - de namen van eenheden zoals variabelen en functies - herkend en tijdelijk opgeslagen voor de parser. De parser controleert tijdens de analyse van de syntax of de naam al in de symbolentabel is opgenomen. Afhankelijk van de grammaticale toestand waarin de parser zich bevindt, wordt een naam opgeslagen, opgevraagd of ongeldig verklaard.

Ook in andere fasen van het vertalen wordt de symbolentabel geraadpleegd. Ook tijdens het optimaliseren en genereren van de machinetaal is informatie over het type, de opslagwijze, de startwaarde, de argumenten en de afmetingen van een naam nodig.

Variabelen en functies die op een stapel gecreëerd zijn, hebben een beperkte levensduur, de naam kan weer gebruikt worden voor nieuwe variabelen en functies. Daarnaast zijn er programmeertalen waarin een naam in verschillende programma-eenheden hergebruikt mag worden, mits dat geen konflikten of dubbelzinnigheden veroorzaakt,

In sommige programmeertalen is het mogelijk 'functions' (zonder neveneffecten) en 'procedures' (met neveneffecten), dezelfde naam te geven als een variabele.

De scope van de naam is beperkt tot programma-eenheden zoals ‘functions’, ‘procedures’, ‘blocks’ ($\{\ldots\}$), ‘modules’, ‘classes’ of ‘namespaces’. Globale namen zijn per definitie overal geldig.

Om snel te kunnen opvragen en wijzigen en onafhankelijk te zijn van het aantal identifiers, wordt een symbolentabel vaak als een hashtabel gerealiseerd.

4.9.2 Elementaire variabelen

Tot nu toe zijn in de voorbeelden elementaire variabelen gebruikt met een onbekend afmeting in bits. Het reserveren van zo’n eenvoudige variabele kostte maar één geheugencel. In echte programma’s kunnen absolute- en integervariabelen van 8, 16, 32 en 64 bits (zie app. E) voorkomen. Ook zijn er variabelen die rationele waarden bevatten van het type float. Deze floats hebben een lengte van 32, 64 of 80 bits (zie app. F). Float berekeningen worden vaak met speciale registers en/of stapelinstructies uitgevoerd:

instructie	werking
$fld\ f$	leg constante op de stapel
$fld\ (f)$	leg inhoud op de stapel
$fst\ (f)$	neem inhoud van de stapel
$fadd$	optelling met de stapel
$fsub$	aftrekking met de stapel
$fmul$	vermenigvuldiging met de stapel
$fdiv$	deling met de stapel

Tenslotte zijn er niet-numerieke variabelen zoals het type ‘char’, ‘byte’ en ‘string’. Een statische string is een rij karakters in opeenvolgende geheugencellen, afgesloten met een ‘nul-karakter’. Dynamische strings worden vaak gerealiseerd met wijzers.

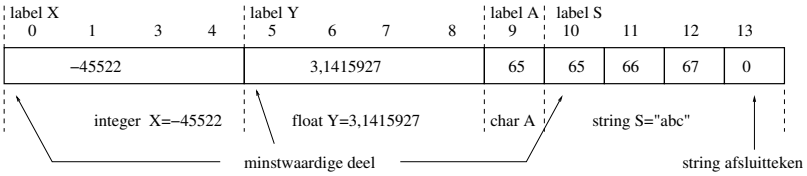
Om elementaire numerieke- en niet-numerieke variabelen op te slaan zijn de volgende methoden mogelijk:

volgorde	principe	label	voorbeelden
Little Endian	minstwaardige deel eerst	eerste cel	IA-32, Z80, 6502 Alpha, PDP11
Big Endian	meestwaardige deel eerst	eerste cel	68xx(x), 8051 AS-400, Sparc, MIPS
Big&Little Endian	afh. van initialisatie	eerste cel	Power PC, Itanium SGI MIPS
woordgeadresseerd	variabele per geheugencel	deze cel	CDC, IBM 7xxx oudere CPU’s

De ‘Endian-volgorde’ is ook bij transmissie en opslag van binaire gegevens belangrijk. Internetprotocollen en ‘Java’ werken bijvoorbeeld met ‘Big Endian’. Bestanden en berichten met binaire gegevens in een andere volgorde, moeten eerst in de ‘Big Endian’

volgorde worden omgezet. ‘Little Endian’ heeft daarentegen het voordeel dat de verwerking van getallen overeenkomt met de volgorde bij het optellen en vermenigvuldigen van cijfers in getallen. Ook de tekens in een string worden met dezelfde volgorde in de geheugencellen geplaatst.

Voorbeeld 4.13



Figuur 4.9: Little Endian

De volgorde van de bits in een geheugencel kan op verschillende manieren. Het is echter gebruikelijk de bits in dezelfde Endian-volgorde te plaatsen als de variabelen in de geheugencellen. Echter bits worden meestal met toenemende index, van rechts naar links getekend. Dit in tegenstelling tot de tekens in een string die met toenemende index, van links naar rechts worden getekend.

De adressering van een bit in een geheugencel of register wordt met een masker gedaan. In een register of geheugencel met n bits, wordt het bit $b = 0 \ldots n-1$ gezet met het masker 2^b in de instructie $X \leftarrow X$ *or* (2^b) en afgezet met het masker $2^n - 1 - 2^b$ in de instructie $X \leftarrow X$ *and* $(2^n - 1 - 2^b)$. De meeste machines hebben daar standaard instructies voor.

Voorbeeld 4.14 *In een geheugencel van* 8 *bits, dan het* 3_{de} *bit (geteld als* $0,1,2$ *met de gewichtswaarde* $2^2 = 4$*) gezet met een or instructie: “*`or X,4`*”. Afzetten van dit bit gaat met een and instructie: “*`and X,251`*”.*

4.9.3 Arrays

De meest eenvoudige array, is een statische één-dimensionale rij van geheugencellen $A[n]$ met een vaste afmeting n. De eerste geheugencel heeft de index 0 en de laatste geheugencel een index $n-1$. Het adresseren van een geheugencel met de index x geschiedt met een formule voor het ‘effectieve adres’: $\#A[x] = \#A + x$. Hierbij is $\#A$ het label van de eerste geheugencel. Dit zou voor $A[x] = 12$ de volgende assemblerinstructie opleveren:

```
          ...
          mov   R,#A      ; startadres array A
          add   R,X       ; #A+X
          mov   (R),12    ; A[X]=12
; Of met een 'super-de-luxe' macro:
          mov   (#A+X),12    ; A[X]=12
```

De macro die als eerste parameter het 'effectief adres': "`#A+X`" en als tweede parameter de waarde: "12" mee kreeg, kan meestal rechtstreeks als instructie uitgevoerd worden. Bijvoorbeeld, in de 'IA-32' machinetalen is adressering met meerdere registers, indexering en schaalfactoren mogelijk [37].

Arrays met inclusief gedeclareerde boven- en ondergrenzen $A[l-h]$ voor arrayelementen met een afmeting a, kunnen met $\#A[x] = \#A - l + x * a$ geadresseerd worden. Bijvoorbeeld:

```
; In single float array A[2-10] wordt A[5]=7
          mov   R,#A-2+5*4 ; adres single float A[5]
          mov   (R),7      ; single float A[5]=7
```

Het overschrijden van een arraygrens kan in beperkte mate bewaakt worden door de compiler. Bij een constante index c moet gelden dat $\#A - laag + c \leq hoog$. Bij indexen in variabelen kan deze bewaking niet meer door de compiler uitgeoefend worden. Alleen als er extra bewakingsinstructies worden mee gegenereerd in het doelprogramma, is het alsnog mogelijk. Dit maakt het doelprogramma trager maar wel veiliger.

Meer-dimensionale arrays komen in meerdere smaken. Het is o.a. mogelijk eerst de rijen (row-major) of eerst de kolommen (column-major) op te slaan. Ook kunnen meerdimensionale arrays gerealiseerd worden als lijsten met wijzers.

Voorbeeld 4.15 *De 'row-major' methode:*

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = [(1,2,3),(4,5,6),(7,8,9)]$$

Voor een meerdimensional array in row-major opgeslagen met de inclusieve onder- en bovengrenzen $A[l_0-h_0, l_1-h_1, \ldots, l_{n-1}-h_{n-1}]$, bestaande uit elementen met de afmeting ('schaalfactor') a, wordt het effectieve adres van het element $x_0, x_2, \ldots x_{n-1}$ berekend met de formule:

$$adres(x_0, x_1, \ldots x_{n-1}) = a \cdot \sum_{i=0}^{n-1} \left((x_i - l_i) \cdot \prod_{1 \leq j < i} (h_j - l_j + 1) \right)$$

Voorbeeld 4.16 *Bij C en C++ heeft een element x_0, x_1 in het array "`int A[n][m];`") met de afmetingen $0 \ldots n-1, 0 \ldots m-1$ en een element met de schaalfactor 4, heeft het effectieve adres:*

$$adres(x_0, x_1) = 4 \cdot (x_0 + x_1 \cdot n)$$

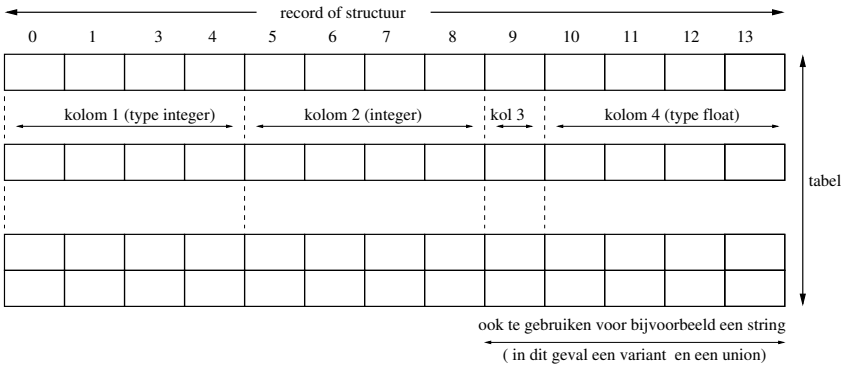
Voor adressering van dynamisch gecreëerde arrays is het startadres in de heap noodzakelijk. Dit adres wordt bij de creatie van het array aangeboden in een register. De inhoud van dit register moet bij het effectieve adres worden opgeteld.

4.9.4 Record-, tabel- en objectstructuren

Een een ‘record’, ookwel ‘lijst’ of ‘structuur’ genoemd, kan verschillende type elementen met verschillende afmetingen hebben. De elementen in een record worden ‘velden’ genoemd. Er bestaan twee soorten records. Records met een vaste lengte en records met variabele lengte. Voor statische recordstructuren moet de symbolentabel de naam, de indexering, en het type van elk veld, bewaren. Records en lijsten met variabele lengte worden meestal dynamisch gerealiseerd met wijzers.

Een één-dimensionele array van records met een vaste lengte, is een tabel. Deze constructie gedraagt zich als een twee-dimensionaal array met rijen en kolommen. Alleen heeft elke kolom een eigen veldbreedte. Statische tabelrijen worden in het algemeen geadresseerd met een adresformule voor een array. Binnen zo’n tabelrij wordt indexering van velden in een record gebruikt.

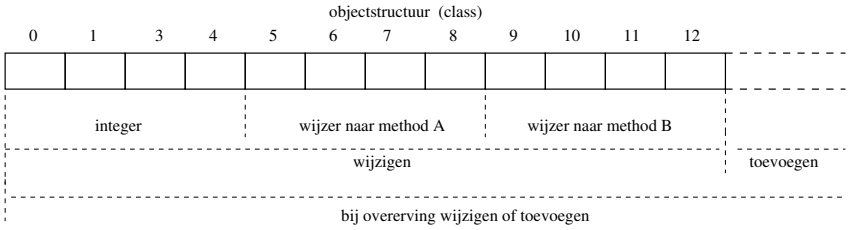
Voorbeeld 4.17



Figuur 4.10: Record- en tabelstructuur

Indien in een record een veld voor meerdere typen elementen gebruikt mag worden, noemt men dat een ‘variant’. Indien meerdere velden gecombineerd mogen worden tot een nieuw veld dan noemt men dat een ‘union’. Indien in record variantie en union is toegestaan en de velden ook adressen van functies en procedures bevatten, dan is er sprake van een ‘objectstructuur’. De typedefinitie van zo’n objectstructuur komt overeen de definitie van een ‘class’ en de declaratie van zo’n objectstructuur met een object. Objecten worden niet in tabellen ondergebracht, maar dynamisch gecreëerd in de heap en geadresseerd met wijzers. Indien een object niet meer nodig is, wordt zij vernietigd en worden de vrijgekomen geheugencellen weer in de vrije ruimte van de heap opgenomen. Programma’s geschreven in ‘Java’, ‘Perl’, PHP en ‘Eiffel’ hoeven dit niet zelf te doen, het wordt automatisch gedaan door een garbage collector. Maar programma’s geschreven in ‘C++’, moeten zelf voor de vrije ruimte zorgen. Een garbage collector voorkomt ‘memory leaks’, maar kan daarentegen wel vertragend op de uitvoering van het programma werken.

Voorbeeld 4.18



Figuur 4.11: Een rudimentaire objectstructuur

4.10 Opgaven

1. Waarom behoren de macro-preprocessor en de macro-postprocessor tot de externe fasen van een compiler?
2. Wat zou het voor- en nadeel van een assemblerfase voor een compiler zijn?
3. Definieer de macro "`for(#1;#2;#3) #4;`" met behulp van een "`while`".
4. Geef een flex programma voor een gebroken getal (float).
5. Geef een adresformule met de registers van de IA-32-architectuur voor een tweedimensionaal array $A[0-n,0-m]$ (app. G).
6. Geef de formule voor het effectieve adres van elementen in arrays met de 'column-major' volgorde.
7. Geef de bisonversie voor: $E \rightarrow (E+T, E-T, T)$, $T \rightarrow (T*F, T/F, F)$, $F \rightarrow (n, i, -E, (E))$.
8. Vertaal $y = 4*x/(x^2+x-1)$ in
   (a) de beschreven stapelgeoriënteerde machinetaal;
   (b) de beschreven registergeoriënteerde machinetaal;
9. Hoe kan men "`while(i<10)`" in sprong- en test-instructies uitdrukken?
10. Hoe kan men "`for(i=0,i<10;i++)`" in sprong- en test-instructies uitdrukken?
11. Wat zijn de belangrijkste verschillen tussen 'JVM' en 'CLI'? Verklaar de consequenties van beide architecturen.

Bijlage A

De ASCII-karaktertabel

oct	dec	hex	kar	oct	dec	hex	kar	oct	dec	hex	kar	oct	dec	hex	kar
000	0	00	NUL	040	32	20		100	64	40	@	140	96	60	‘
001	1	01	SOH	041	33	21	!	101	65	41	A	141	97	61	a
002	2	02	STX	042	34	22	”	102	66	42	B	142	98	62	b
003	3	03	ETX	043	35	23	#	103	67	43	C	143	99	63	c
004	4	04	EOT	044	36	24	$	104	68	44	D	144	100	64	d
005	5	05	ENQ	045	37	25	%	105	69	45	E	145	101	65	e
006	6	06	ACK	046	38	26	&	106	70	46	F	146	102	66	f
007	7	07	BEL	047	39	27	’	107	71	47	G	147	103	67	g
010	8	08	BS	050	40	28	(	110	72	48	H	150	104	68	h
011	9	09	HT	051	41	29	)	111	73	49	I	151	105	69	i
012	10	0A	LF	052	42	2A	*	112	74	4A	J	152	106	6A	j
013	11	0B	VT	053	43	2B	+	113	75	4B	K	153	107	6B	k
014	12	0C	FF	054	44	2C	,	114	76	4C	L	154	108	6C	l
015	13	0D	CR	055	45	2D	-	115	77	4D	M	155	109	6D	m
016	14	0E	SO	056	46	2E	.	116	78	4E	N	156	110	6E	n
017	15	0F	SI	057	47	2F	/	117	79	4F	O	157	111	6F	o
020	16	10	DLE	060	48	30	0	120	80	50	P	160	112	70	p
021	17	11	DC1	061	49	31	1	121	81	51	Q	161	113	71	q
022	18	12	DC2	062	50	32	2	122	82	52	R	162	114	72	r
023	19	13	DC3	063	51	33	3	123	83	53	S	163	115	73	s
024	20	14	DC4	064	52	34	4	124	84	54	T	164	116	74	t
025	21	15	NAK	065	53	35	5	125	85	55	U	165	117	75	u
026	22	16	SYN	066	54	36	6	126	86	56	V	166	118	76	v
027	23	17	ETB	067	55	37	7	127	87	57	W	167	119	77	w
030	24	18	CAN	070	56	38	8	130	88	58	X	170	120	78	x
031	25	19	EM	071	57	39	9	131	89	59	Y	171	121	79	y
032	26	1A	SUB	072	58	3A	:	132	90	5A	Z	172	122	7A	z
033	27	1B	ESC	073	59	3B	;	133	91	5B	[	173	123	7B	{
034	28	1C	FS	074	60	3C	<	134	92	5C	\	174	124	7C	—
035	29	1D	GS	075	61	3D	=	135	93	5D	]	175	125	7D	}
036	30	1E	RS	076	62	3E	>	136	94	5E	^	176	126	7E	~
037	31	1F	US	077	63	3F	?	137	95	5F	_	177	127	7F	DEL

Bijlage B

Reguliere expressies

Symbolen en tekens	
$\backslash 0n, \backslash 0nn, \backslash 0mnn$	octale waarde karakter $00-07$, $000-077$, $0000-0377$
$\backslash xhh$, $\backslash uhhhh$	hexadecimale waarde karakter $0xhh$, $0xhhhh$
$\backslash t$	tab-karakter ('$\backslash u0009$')
$\backslash n$	newline-karakter (line feed) ('$\backslash u000A$')
$\backslash r$	carriagereturn-karakter ('$\backslash u000D$')
$\backslash f$	formfeed-karakter ('$\backslash u000C$')
$\backslash a$	alert-karakter (bell) ('$\backslash u0007$')
$\backslash e$	escape-karakter (escape) ('$\backslash u001B$')
$\backslash cx$	controlekarakter <x>
Verzamelingen en rijen	
[abc]	a, b, or c (opsomming)
[^abc]	alles behalve a, b, or c (complement)
[a-zA-Z]	a tot/met z van A tot/met Z, (inclusieve rij)
[a-d[m-p]]	a tot/met d of van m tot/met p: [a-dm-p](samenvoeging)
[a-z&&[def]]	d, e, of f (gemeenschappelijk)
[a-z&&[^bc]]	a tot/met z, behalve b en c: [ad-z](verwijdering)
[a-z&&[^m-p]]	a tot/met z, behalve m tot/met p: [a-lq-z](verwijdering)
Voorgedefinieerde verzamelingen en rijen	
.	alle karakters (eventueel met linefeed/carriage-return)
$\backslash d$	cijfers: [0-9]
$\backslash D$	niet-cijfers: [^0-9]
$\backslash s$	witte ruimte karakters: [$\backslash t \backslash n \backslash x0B \backslash f \backslash r$]
$\backslash S$	niet-witte ruimte karakters: [^$\backslash s$]
$\backslash w$	woordkarakters (letters/cijfers): [a-zA-Z_0-9]
$\backslash W$	niet-woordkarakters: [^$\backslash w$]
Positiesymbolen	
^	begin van een regel
$	einde van een regel
$\backslash b$	woordgrens
$\backslash B$	niet-woordgrens
$\backslash A$	begin van de invoer
$\backslash G$	einde van de vorige herkenning
$\backslash Z$	einde van de invoer maar voor een eventuele afsluiting
$\backslash z$	einde van de invoer

Gulzige kwantoren	
X?	één of niets
X$\star$	nul of meer
X+	één of meer
X{n}	exact *n* keer
X{n,}	tenminste *n* keer
X{n,m}	tenminste *n* maar niet meer dan *m* keer
Niet-gulzige kwantoren	
X??	één of niets
X$\star$?	nul of meer
X+?	een of meer
X{n}?	exact *n* keer
X{n,}?	tenminste *n* keer
X{n,m}?	tenminste *n* maar niet meer dan *m* keer
Bezitterige kwantoren	
X?+	één of niets
X$\star$+	nul of meer
X++	een of meer
X{n}+	exact *n* keer
X{n,}+	tenminste *n* keer
X{n,m}+	tenminste *n* maar niet meer dan *m* keer
Logische voegwoorden	
XY	patroon X gevolgd door patroon Y
X|Y	patroon X òf patroon Y (alternatief)
(X)	patroon X gegroepeerd
Groepering	
$\backslash n$	gegroepeerde patronen $\backslash 0 \ldots \backslash 9$. In Perl: $\backslash 10$ en hoger mogelijk
Escapen	
$\backslash$	escape-karakter
$\backslash l$	maakt een kleine letter van het volgende teken
$\backslash u$	maakt een hoofdletter van het volgende teken
$\backslash L$	maakt kleine letters (tot $\backslash E$)
$\backslash U$	maakt hoofdletters (tot $\backslash E$)
$\backslash Q$	begin escape (alles van $\backslash Q$ tot $\backslash E$)
$\backslash E$	einde escape
Positieve en negatieve heen- en terugverwijzingen	
(? :X)	group X
(? =X)	positieve lookahead van groep X
(?!X)	negatieve lookahead van groep X
(? <=X)	positieve lookbehind van groep X
(? <!X)	negatieve lookbehind van groep X
(? >X)	groep X doet niet mee aan herkenning

POSIX verzamelingen. In Perl met [:Lower:] notatie.	
\p{Lower}, [:Lower:]	kleine letters: [a-z]
\p{Upper}, [:Upper:]	hoofdletters: [A-Z]
\p{ASCII}, [:ASCII:]	alle ASCII's: [\x00-\x7F]
\p{Alpha}, [:Alpha:]	alphabetische karakters: [\p{Lower}\p{Upper}]
\p{Digit}, [:Digit:]	decimale cijfers: [0-9]
\p{Alnum}, [:Alnum:]	alphanumerieken: [\p{Alpha}\p{Digit}]
\p{Punct}, [:Punct:]	leestekens: !"#$%&'()*+,-./:;<=>?@[\]^_`{|}
\p{Graph}, [:Graph:]	leesbare karakters: [\p{Alnum}\p{Punct}]
\p{Print}, [:Print:]	printbare karakters: [\p{Graph}]
\p{Blank}, [:Blank:]	spatie of tab: [\t]
\p{Cntrl}, [:Cntrl:]	controlekarakters: [\x00-\x1F\x7F]
\p{XDigit}, [:XDigit:]	hexadecimale cijfers: [0-9a-fA-F]
\p{Space}, [:Space:]	witte ruimte: [\t\n\x0B\f\r]
Unicode blokken	
\p{InGreek}	Griekse karakters (eenvoudige verzameling)
\p{Lu}	hoofdletter (eenvoudige verzameling)
\p{Sc}	valutatekens
\P{InGreek}	alle andere karakters dan de Griekse (complement)
[\p{L}&&[^\p{Lu}]	alle letters behalve hoofdletters (verwijdering)

Unicode blokken (verzamelingen van niet-latijnse karakters) worden met \p en \P aangegeven. De constructie \p{blok} geeft aan dat de speciale karakters in het patroon herkend worden. Daarentegen geeft de constructie \P{blok} aan dat de speciale karakters niet herkend worden. De blokken zijn gedefinieerd in de Unicode Standaard, Versie 3.0.

Argumenten

Tenslotte de argumenten die het patroonherkennen beïnvloeden:

Argumenten	
c	bereidt voor op voortzetting (in combinatie met g).
g	zoek zoveel mogelijk naar het patroon.
i	hoofdletter gevoelig.
o	hoofdletter ongevoelig.
m	behandel string als meerdere regels, ^ en $ refereren aan \n of \r\n.
s	behandel string als één regel.
x	spatie en commentaren toegestaan.

Bijlage C

Verschillen tussen reguliere expressies

symbool	emacs	vi	sed	awk	grep	Perl	action
\	√	√	√	√	√	√	escapesymbool
^	√	√	√	√	√	√	begin regel
$	√	√	√	√	√	√	einde regel
\ < \ >	√	√					begin/einde woord
[ ]	√	√	√	√	√	√	verzameling
[^]	√	√	√	√	√	√	complement verzameling
[-]	√	√	√	√	√	√	rij
\(\)	√		√				groep
( )				√		√	groep
.	√	√	√	√	√	√	elk karakter (wildcard)
*	√	√	√	√	√	√	nul of meer (kwantor)
+	√			√	√	√	één of meer (kwantor)
?				√	√	√	nul of één (kwantor)
{ , }				√	√	√	numerieke (kwantor)
\{,\}			√				numerieke (kwantor)
—				√		√	alternatieven

Bijlage D

grep, sed, awk en Perl

grep

Het programma grep zoekt patronen in bestanden of rechtstreekse invoer. Zodra een patroon herkend is, print grep de gehele regel uit waarin dit patroon gevonden is.

```
grep [options] patroom [file(s)]
```

Grep wordt vaak gebruikt als een filter in een pipe.

```
ps -ax|grep .*term|grep peter
```

sed

Als filter (let op de andere naam van de uitvoer):

```
sed -e  's/funktie/functie/g' <hfst1.tex >hfst1a.tex
```

of

```
cat hfst1.tex | sed -e 's/funktie/functie/g' >hfst1a.tex
```

met een script:

```
cat hfst1.tex | sed -f sedscr >hfst1a.tex
```

waarbij sedsrc de volgende inhoud heeft:

```
s/funktie/functie/g
s/konstant/constant/g
```

awk

- Een awk script leest regel voor regel in;
- BEGIN-END blokken worden buiten deze inlees-lus geplaatst;
- Een awk instructie is in principe een combinatie van een patroon met een actie, zo'n actie kan in principe alles zijn wat op C lijkt, zelfs printopdrachten.

```
#Dit awk-script controleert of er lege regels leeg zijn
#of dat er getallen of teksten aanwezig zijn.
/[0-9]+/ { print "Een nummer gevonden" }
/[a-zA-Z]+/ { print "Een string gevonden" }
/^$/ { print "Een lege regel gevonden " }
```

Perl

Perl als filter om hoofd- in kleine letters te veranderen:

```
echo "Hallo Wereld" | perl -pe 's/([^\W0-9_])/\l\1/g'
>hallo wereld
```

Een geavanceerder voorbeeld is het oplossen van de geheeltallige vergelijking $2 \cdot x + 3 \cdot y + 5 \cdot z = 171$ voor $0 \leq x$, $0 \leq y$ en $0 \leq z$ met behulp van het backtracken [6]:

```
#!/usr/bin/perl
if (($X, $Y, $Z) =
    (('1' x 171)  =~ /^(1*)\1{1}(1*)\2{2}(1*)\3{4}$/))
{
    ($x, $y, $z) = (length($X),length($Y),length($Z));
    print "een oplossing :  2*$x + 3*$y + 5*$z = 171.\n";
} else {
    print "geen oplossing.\n";
}
>een oplossing :  2*84 + 3*1 + 5*0 = 171.
```

Als het gedrag van kwantoren wordt veranderd, zijn andere oplossingen mogelijk.

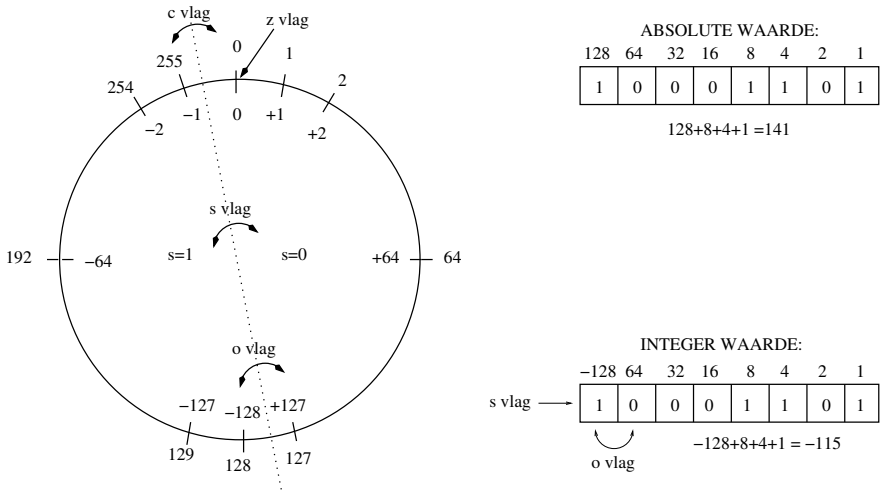
Een andere rekenkundige toepassing van een reguliere expressie met backtracking is een test of een getal een priemgetal is of niet:

```
>echo 1009|perl -pe 's/(\d+)/"1" x $1/e'| \
          perl -pe 's/^1?$|^(11+?)\1+$/niet priem/'| \
          perl -pe 's/1+/priem/'
>priem
>echo 1008|perl -pe 's/(\d+)/"1" x $1/e'| \
          perl -pe 's/^1?$|^(11+?)\1+$/niet priem/'| \
          perl -pe 's/1+/priem/'
>niet priem
```

De regel `perl -pe 's/(\d+)/"1" x $1/e'` maakt van het getal *n* (`echo n`) een lijst met *n* '1'-en. Als in de tweede regel de terugverwijzing `\1` - overeenkomende met `(11+?)` - een geheel aantal keren afgepast kan worden over de rest van de rij met '1'-en, dan is het getal *n* een veelvoud van `11+?`. Door backtracking worden de getallen 2,3,4,5,... uitgeprobeerd als delers `11+?`. Als het backtracken zonder matching stopt, dan is het getal *n* een priemgetal.

Bijlage E

Statusvlaggen in een CPU



Figuur E.1: Vlaggen

Bij een nulwaarde wordt de zero-vlag gezet $z = 1$ anders wordt de z-vlag afgezet $z = 0$. De z-vlag geldt bij absolute- en integerwaarden.

Een absolute variabele met 8 bits, een byte, heeft een waarde tussen $0 \ldots 255$. Absolute variabelen kunnen gebruikt worden als indexen in loops en arrays, pointers en adressen. Als een byte van waarde verandert waarbij de $255 \leftrightarrow 0$ overgang wordt gepasseerd, dan wordt de carry-vlag gezet $c = 1$. Bij alle andere overgangen wordt $c = 0$.

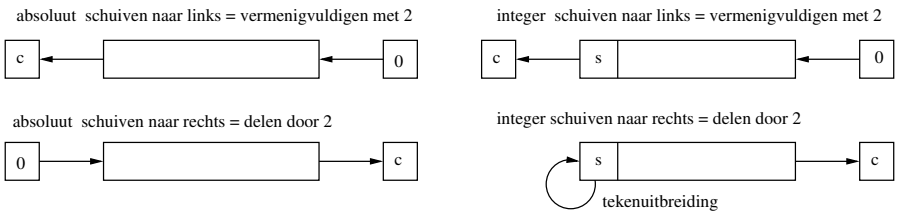
Een integer met 8 bits heeft een waarde tussen $-128 \ldots +127$. Als bij een verandering $+127 \leftrightarrow -128$ wordt gepasseerd, dan wordt de overflow-vlag gezet $o = 1$. In alle andere gevallen wordt $o = 0$. Bij een negatieve waarde van de integer $-128 \ldots -1$ is de sign-vlag gezet $s = 1$. Bij alle andere waarden is $s = 0$.

Bij integer met 16 bits heeft de carry-overgang bij $65535 \leftrightarrow 0$ en de overflow-overgang bij $+32767 \leftrightarrow -32768$. Bij een integer met 32 bits ligt de carry-overgang

bij $42949667295 \leftrightarrow 0$ en de overflow-overgang bij $+2147483647 \leftrightarrow -2147483648$.
De conditionele sprongen zijn gescheiden in absolute en integer condities:

test	absoluut		integer	
	vlaggen	instructie	vlaggen	instructie
$getal = 0$	$z = 1$	je	$z = 1$	je
$getal \neq 0$	$z = 0$	jne	$z = 0$	jne
$getal < 0$	$c = 1$	jb	$o \neq s$	jl
$getal \leq 0$	$c = 1 \vee z = 1$	jbe	$o \neq s \wedge z = 1$	jle
$getal > 0$	$c = 0 \wedge z = 0$	ja	$o = s \wedge z = 0$	jg
$getal \geq 0$	$c = 0$	jae	$o = s$	jge

Ook met het schuiven (delen en vermenigvuldigen met 2) moet men rekening houden met de verschillen tussen absolute en integerwaarden:

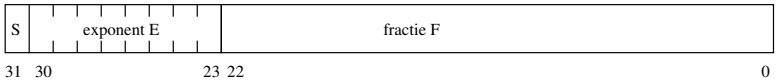


Figuur E.2: Schuifacties met absolute- en integerwaarden

Bijlage F

Floating Point (IEEE Standard 754)

float	lengte	E	F	bereik	
type	bit			binair	decimaal
single	32	8	23	$2^{-126}\ldots2^{127}$	$1,18\cdot10^{-38}\ldots3,40\cdot10^{38}$
double	64	11	52	$2^{-1022}\ldots2^{1023}$	$2,23\cdot10^{-308}\ldots1,79\cdot10^{308}$
extended	80	15	64	$2^{-16382}\ldots2^{16383}$	$3,37\cdot10^{-4932}\ldots1,18\cdot10^{4932}$



Figuur F.1: Single Precision Float

De lengte van een 'single float' is $S+E+F=1+8+23=32$. S is het 'tekenbit', de absolute waarde E is de 'exponent met bias', de integerwaarde $E-127$ is de 'exponent', de absolute waarde 127 is de 'bias' en de absolute waarde F is de 'fractie'. De bias is nodig om negatieve exponenten te kunnen maken.

Voor een 'single float' zijn de maximale waarden voor $E=0\ldots255$ en voor $F=0\ldots8388607$. De ingewikkelde formule voor de waarde $w=(-1)^S\cdot2^{E-B}\cdot1,F$ van een float, wordt vereenvoudigd tot $w=\pm2^e\cdot m$ met de mantisse $m=1,f$ en de exponent $e=E-B$.

operatie	resultaat
f_1*f_2	$2^{e_1+e_2}\cdot m_1\cdot m_2$
f_1/f_2	$2^{e_1-e_2}\cdot m_1/m_2$
$f_1\pm f_2$	$2^e\cdot(m_1\pm m_2)$ als $e_1=e_2=e$

Voordat twee floats worden opgeteld of afgetrokken, moeten de exponenten gelijk zijn. Daarom wordt de gemeenschappelijke exponent $e'=0$. Hierdoor worden de nieuwe mantissen m'_1 en m'_2 van beide getallen ongenormaliseerd: $m'_x=m/2^{0-e_x}=0,f_x/2^{-e_x}$. Na de

optelling of aftrekking, moet de mantisse van het resultaat weer genormaliseerd worden tot $m_r = 1, f$. De leidende nullen uit de fractie worden door herhaald vermenigvuldigen met 2 aan de meestwaardige kant eruit geschoven en de exponent wordt daarbij telkens met 1 verhoogd, totdat $m'_r = 1, f'_r$.

Float-operaties kunnen tot de volgende resultaten leiden:

operaties $(w \neq 0)$	resultaat
$\pm w / \pm \infty$	0
$\pm w / 0$	$\pm\infty$
$\pm\infty \cdot \pm\infty$	$\pm\infty$
$\infty + \infty$	∞
$\infty - \infty$	NaN
$\pm\infty / \pm\infty$	NaN
$\pm\infty \cdot 0$	NaN
$\pm 0 / \pm 0$	NaN

Een foutief resultaat is een 'Not a Number' (NaN). Dit kan een 'Quiet Not a Number' $(QNaN)$ of een 'Signaling Not a Number' $(SNaN)$ zijn. De $QNaN$ waarden ontstaan bij onbepaalde resultaten en mogen wel verder gebruikt worden, $SNaN$ waarden ontstaan door illegale operaties en mogen niet meer gebruikt worden. De genormaliseerde-, de ongenormaliseerde en de NaN floatwaarden zijn voor $E = 0 \ldots E_{max}$ en $F = 0 \ldots F_{max}$ gedefinieerd als:

waarde	S	E	F	verklaring
positief nul	0	0	0	$+0$
negatief nul	1	0	0	-0
positief ongenormaliseerd	0	0	$\neq 0$	$+2^{-B+1} \cdot 0, f$
negatief ongenormaliseerd	1	0	$\neq 0$	$-2^{-B+1} \cdot 0, f$
positief genormaliseerd	0	$0 < E < E_{max}$	$0 \leq F \leq F_{max}$	$+ \cdot 2^{E-B} \cdot 1, f$
negatief genormaliseerd	1	$0 < E < E_{max}$	$0 \leq F \leq F_{max}$	$- \cdot 2^{E-B} \cdot 1, f$
positief oneindig	0	$E = E_{max}$	0	$+\infty$
negatief oneindig	1	$E = E_{max}$	0	$-\infty$
$QNaN$	0	$E = E_{max}$	$F > F_{max}/2$	Quiet Not a Number
$QNaN$	1	$E = E_{max}$	$F > F_{max}/2$	Quiet Not a Number
$SNaN$	0	$E = E_{max}$	$F \leq F_{max}/2$	Signalling Not a Number
$SNaN$	1	$E = E_{max}$	$F \leq F_{max}/2$	Signalling Not a Number

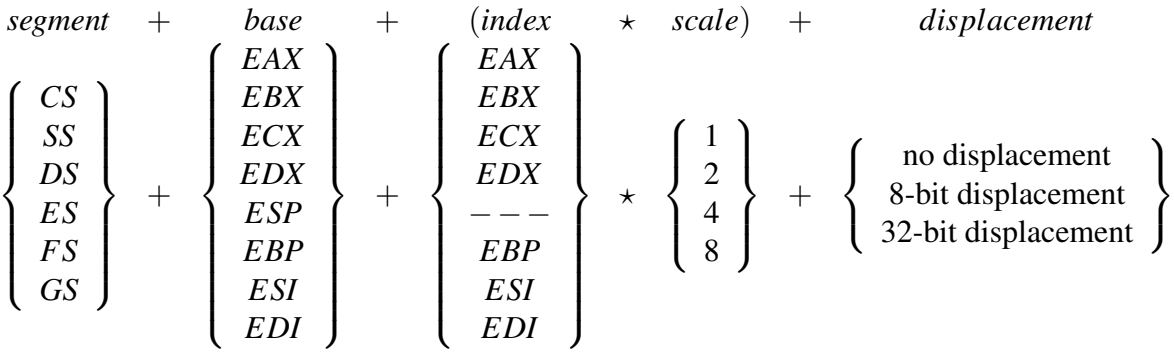
Bijlage G

Intel IA-32, registers en adressering

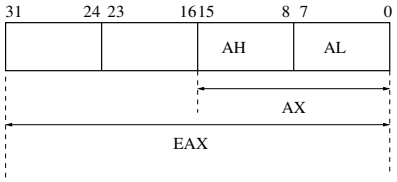
De registers *CS*, *SS*, *DS*, *ES*, *FS* en *GS* zijn segmentregisters. Deze registers verdelen het geheugen is segmenten voor instructies (het code-segment), voor de stapel (het stapel-segment), voor de heap (data-segment) en extra data-segmenten. De instructiewijzer *EIP* wordt altijd aan het code-segment gekoppeld. Andere registers zoals de algemene registers worden standaard aan een segmentregister gekoppeld, tenzij expliciet een ander segment wordt aangegeven, zoals bijvoorbeeld “ `mov FS:EDI,variabele`”.

algemeen register		standaard gekoppeld	
naam	functie	naam	functie
EIP	instructiewijzer	*CS*	code-segment register
ESP	stapelwijzer	*SS*	stapel-segment register
EBP	index stapelwijzer	*SS*	stapel-segment register
EAX	algemeen register	*DS*	data-segment register
EBX	algemeen register	*DS*	data-segment register
ECX	algemeen register	*DS*	data-segment register
EDX	algemeen register	*DS*	data-segment register
ESI	algemeen register	*DS*	data-segment register
EDI	algemeen register	*DS*	data-segment register
		ES FS GS	extra data-segment registers
	string-instructies	*DS* : *ESI*	
	string-instructies	*ES* : *EDI*	

Registers zijn 32 bits als zij in de instructies met hun volledige naam aangeroepen worden. Bijvoorbeeld het 32-bit register *EAX* kan aangeroepen worden als het 8-bit register *AL* of het 16-bit register *AX*. Het meestwaardige deel van het *AX* register, het 8-bit register *AH* is eveneens adresseerbaar. In de IA-64 architectectuur zijn ook 64-bit registers aanwezig.



Figuur G.1: Adresseringsmogelijkheden voor registers



Figuur G.2: Het algemene register *EAX*

adressering	voorbeeld
register naar register	`mov EAX,EBX`
register naar geheugen	`mov VAR,AL`
geheugen naar register	`mov CX,VAR`
constante naar register	`mov CX,3`
constante naar geheugen	`mov VAR,3`
uitgebreide adressering	`mov ES:[EBX+EAX*2]+4,FS:[ECX]+422001`

Literatuur

[1] Alfred V. Aho and Jeffrey D. Ullman: *Principles of Compiler Design*, Addison-Wesley, Reading, MA; 1977

[2] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman: *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA; 1986.

[3] Andrew W. Appel, Maia Ginsburg: *Modern Compiler Implementation in C*,Cambridge University Press, Cambridge, 1998

[4] Andrew W. Appel: *Modern Compiler Implementation in Java*,Cambridge University Press, Cambridge, 1998

[5] Dan Appleman, Daniel Appleman: *Regular Expressions with .NET*, O'Reilly, Cambridge, MA; 2003

[6] Tom Christiansen, Nathan Torkingson: *Perl Cookbook*, O'Reilly, Cambridge, MA; 1998

[7] Jeffrey E. F. Friedl: *Mastering Regular Expressions*, O'Reilly, Cambridge, MA; 2003

[8] Mehran Habibi: *Java Regular Expressions*, Apress; 2003

[9] Seymour Lipschutz, Marc Lipson: *Discrete Mathematics*, Schaum's Outlines Series, McGraw-Hill, NY; 1997

[10] Emil Post: *A variant of a recursively unsolvable problem*, Bull. of Amer. Math. Soc., 52:264*268, 1946.

[11] Peter H. Salus e.a.: *Handbook of Programming Languages VOL III: Little Languages and Tools*, Macmillan Technical Publishing, Indianapolis, IN; 1998

[12] Robert Sedgewick: *Algorithms*, Addison-Wesley, Reading, MA; 1983

[13] Ellen Siever: *Linux in a Nutshell*, O'Reilly, Cambridge, MA; 1999

[14] Niklaus Wirth: *Algorithms+DataStructures=Programs*, Prentice-Hall, Englewood Cliffs, NJ; 1976

Andere bronnen

[15] http://directory.fsf.org/GNU/

[16] http://www.gnu.org/software/m4/m4.html

[17] http://www.perl.org/

[18] http://www.python.org/

[19] http://www.php.net/

[20] http://www.tcl.tk/

[21] http://www.regexlib.com/DisplayPatterns.aspx

[22] http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html

[23] http://ftp.gnu.org/pub/gnu/regex/

[24] http://www.lysator.liu.se/c/ANSI-C-grammar-l.html

[25] http://www.gnu.org/software/flex/manual/

[26] http://www.cs.princeton.edu/ appel/modern/java/JLex/

[27] http://www.lysator.liu.se/c/ANSI-C-grammar-y.html

[28] http://www.gnu.org/software/bison/manual/

[29] http://www.cs.princeton.edu/ appel/modern/java/CUP/

[30] http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html

[31] http://www.ecma-international.org/standards/ecma-335/Ecma-335-part-i-iv.pdf

[32] http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-335.pdf

[33] http://gcc.gnu.org/

[34] http://gcc.gnu.org/onlinedocs/gccint/File-Framework.html

[35] http://nasm.sourceforge.net/wakka.php?wakka=HomePage

[36] http://gcc.gnu.org/onlinedocs/gccint/RTL.html

[37] ftp://download.intel.com/design/Pentium4/manuals/25366514.pdf

[38] http://www.free-definition.com/IEEE-floating-point-standard.html

Index