FACULTEIT INGENIEURSWETENSCHAPPEN
EN ARCHITECTUUR

Compilers - Voorbeeldexamen

Bert De Saffel

Master in de Industriële Wetenschappen: Informatica Academiejaar 2018–2019

Gecompileerd op 11 juni 2019

UNIVERSITEIT
GENT

Inhoudsopgave

Hoofdstuk 1

Vragen

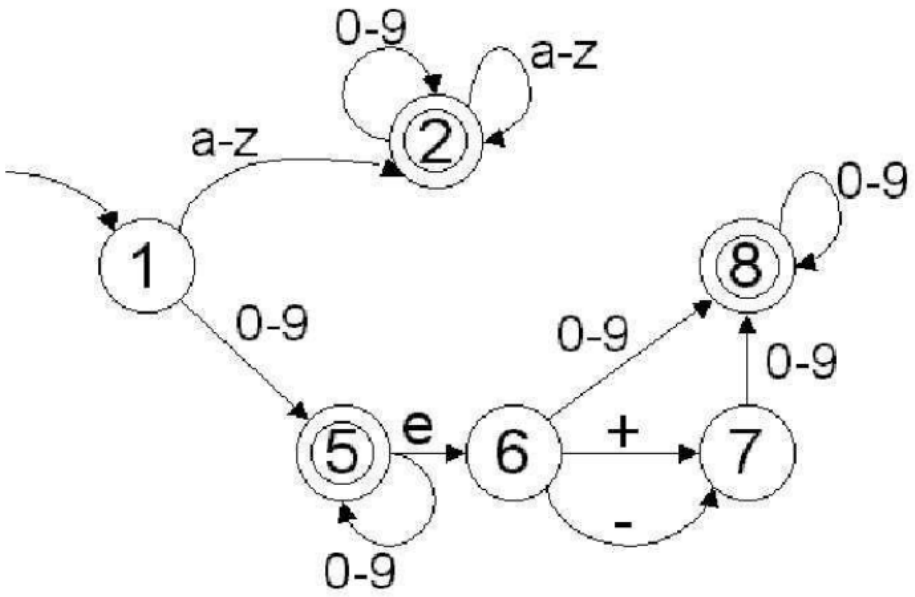
1.1 Grammatica

Gegeven de taal $L = \{a^m b^n c^p\}$ met $m > p$.

1. Kan een reguliere expressie gevormd worden die de woorden uit deze taal herkent? Waarom wel/niet?
2. Maak een niet-ambigue grammatica voor taal L.
3. Kan de taal uitgebreidt worden met de woorden waarvoor geldt $m < p$ en $m \neq p$? Geef de aangepaste grammatica.

Antwoord

1. xd



1.2 Deterministische Eindige Automaten

Herschrijf de volgende deterministische eindige automaat als een reguliere expressie.

```
[a-z][0-9a-z]*|[0-9]+(e[0-9|+|-][0-9]*)+
```

1.3 Compilerfasen - I

Gegeven de opeenvolgende fasen van een compiler, geef aan de programmavoorstellingen, grafen of datastructuren die aan het *eind* van elke fase geproduceerd worden. Geef uw antwoord in de vorm: a-7, b-3, ...

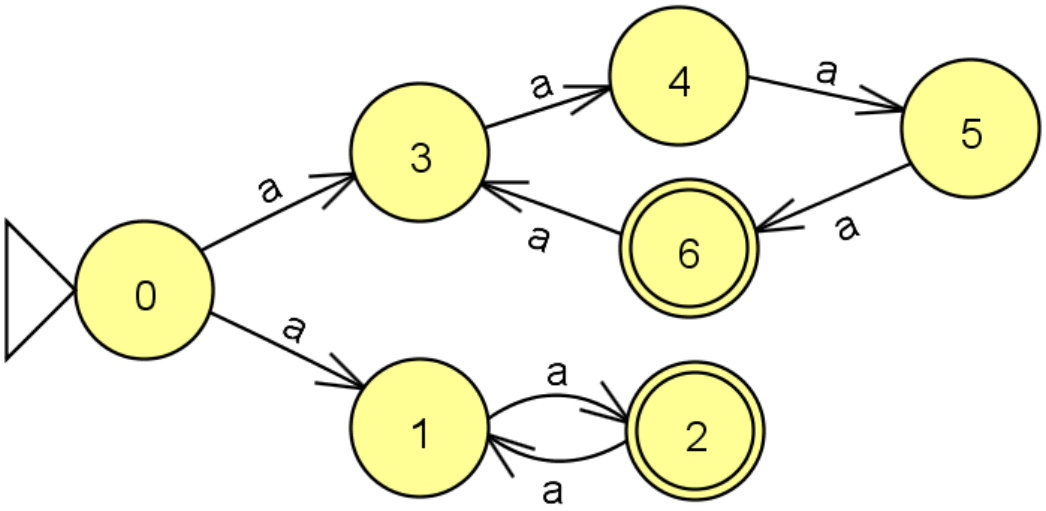
Compilerfase	Programmavoorstelling
(a) scanning (lexicale analyse)	1. intermediaire boomtaal
(b) parsing (syntactische analyse)	2. control flow graaf (CFG)
(c) type checking (semantische analyse)	3. frame layout, activation records
(d) vertaling (translate)	4. symbooltabellen
(e) canonicalisering	5. interferentiegraaf
(f) instructieselectie	6. gekleurde interference graph
(g) controlestroom analyse	7. tokens
(h) dataflow analyse (liveness)	8. assembler code
(i) register allocatie	9. assembler instructies
(j) code emission	10. abstracte syntax tree (AST)

Antwoord

a-7, b-10, c-4, d-3, e-1, f-9, g-2, h-5, i-6, j-8

1.4 Lexicale analyse

Volgende nondeterministische eindige automaat herkent geldige tokens `(aa)+` en `(aaa)+`.



1. Beschrijf de manier om in het algemeen verschillende NFA's samen te voegen tot 1 DFA. Bespreek daarbij de definitie van *sluiting*, *DFAedge* en het *algoritme* dat de NFA-toestanden samenvoegt tot toestanden in een DFA.
2. Pas dit toe op dit voorbeeld: de herkenning van twee- en drievouden van `a` en teken de bekomen DFA.
3. Zijn er in de gevonden DFA nog equivalente toestanden die men kan vereenvoudigen?

Antwoord

1. Een deterministische eindige automaat (DFA) is een eindige automaat waarbij elke transitie uniek is. Het algoritme om een NFA om te vormen naar een DFA maakt enerzijds gebruik van sluitingen. De sluiting T van een verzameling van toestanden S, of `closure(S)`, bevat alle toestanden die kunnen bereikt worden voor de lege transitie ϵ voor elke staat in S. Dit kan wiskundig gedefinieerd worden als:

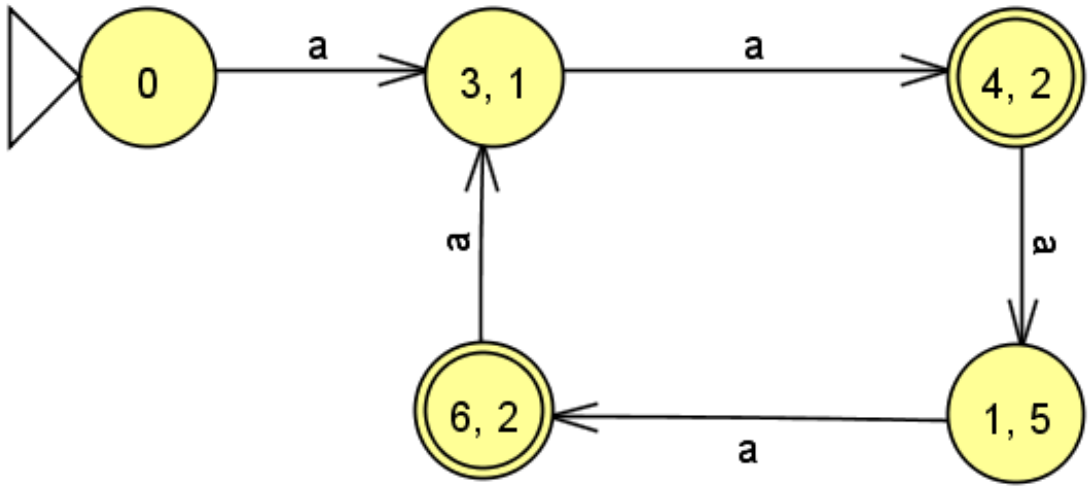
   $$T = S \cup \left(\bigcup_{s \in T} edge(s, \epsilon) \right)$$

   waarbij `edge(s, c)` de staten geeft die vanuit s via symbool c kan bereikt worden.

   Anderzijds maakt het algoritme ook gebruik van de functie `DFAedge(d, c)`, die de staten teruggeeft die vanuit d kunnen bereikt worden bij symbool c.

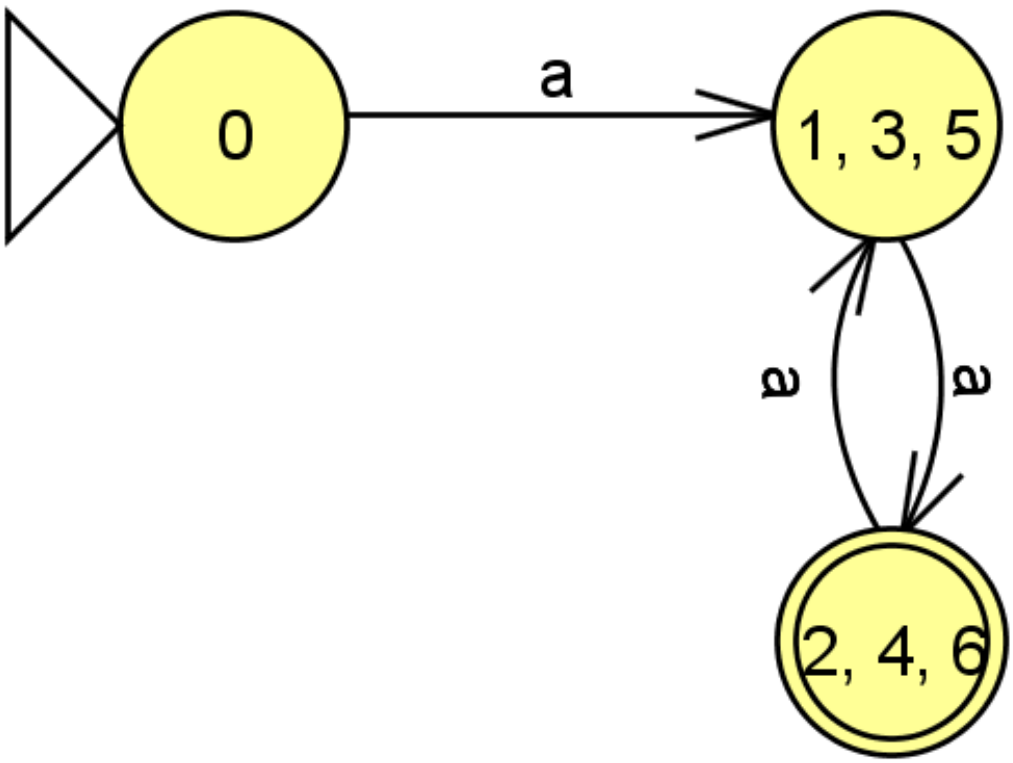
   _ToDo: verder uitleggen

2. De automaat is enkel in staat om enkelvoudige a symbolen te verwerken en aangezien er geen lege strings zijn, draagt de `closure(S)` functie hier niets bij. De eerste verwerking zorgt ervoor dat staten 1 en 3 bereikt worden. Vanuit 1 kan er naar 2 gegaan worden en vanuit 3 kan er naar 4 gegaan worden. In dit geval moet dit herhaald worden tot dat er een combinatie van staten gevonden is die al eerder gecombineerd zijn. Dit is het geval als uiteindelijk de staten



6 en 2 bereikt worden. Vanuit 6 kan naar 3 gegaan worden en vanuit 2 kan naar 1 gegaan worden. Die combinatie bestaat al, en da DFA is voltooid.

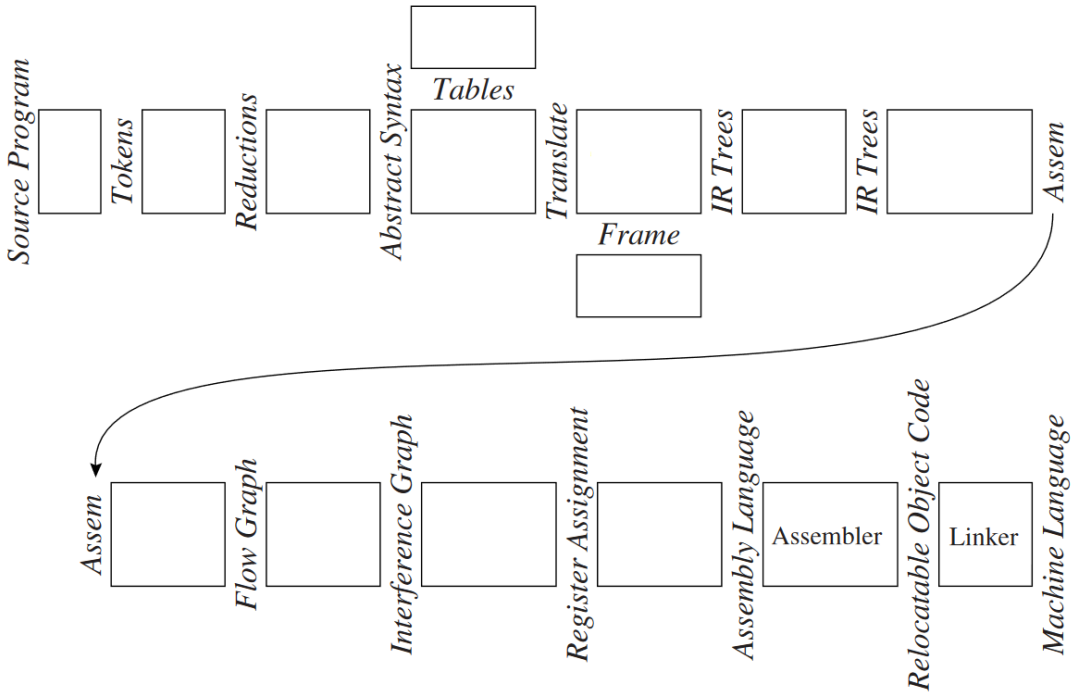
3. Het algoritme garandeerd niet dat de opgeleverde DFA optimaal is, maar er kunnen nadien wel nog optimalisaties doorgevoerd worden. Staten die equivalent zijn kunnen samengenomen worden. Een staat s_1 is equivalent met staat s_2 als ze beiden finaal of niet finaal zijn voor dezelfde symbolen en als voor elk symbool c, $\text{trans}[s_1, c] = \text{trans}[s_1, c]$. In dit geval is dit waar voor staat $\{3, 1\}$ met $\{1, 5\}$ en voor staat $\{4, 2\}$ met $\{6, 2\}$. Deze kunnen gecombineerd worden.



1.5 Compilerfasen - II

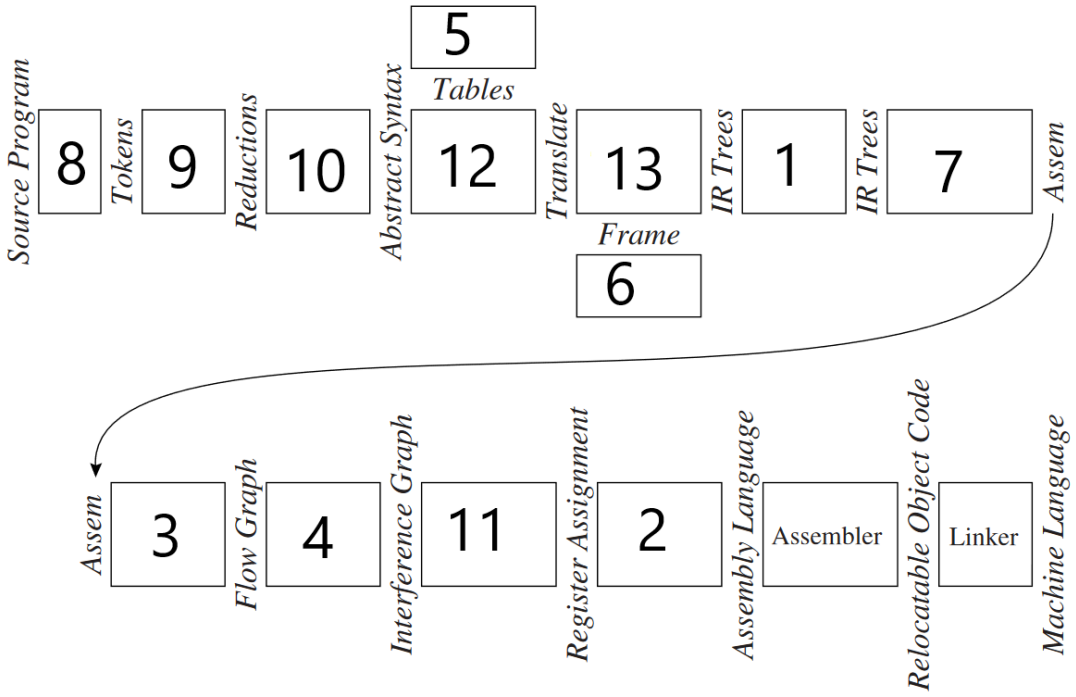
De volgende figuur toont de intermediaire voorstellingen die gebruikt worden als interface tussen de verschillende fasen van een compiler. Vul de nummers in van de corresponderende fasen in een compiler.

1. Canonicalize
2. Code Emission
3. Control Flow Analysis
4. Data Flow Analysis
5. Environments
6. Frame Layout
7. Instruction Selection
8. Lex
9. Parse
10. Parsing Actions



11. Register Allocation
12. Semantic Analysis
13. Translate

Antwoord



1.6 Compilerfasen - III

1. Onderlijn de zo goed als verplichte analyse- en transformatiestappen in een complete compiler in de onderstaande figuur. Onderlijn dus niet de optionele optimalisaties en enkel daarvoor benodigde analyses.
2. Geef met pijlen met volle lijnen de volgorde aan waarin die verplichte stappen normaal uitgevoerd worden.
3. Geef met pijlen met stippelijnen aan in welke volgorde en waar tussen de verplichte stappen de optionele analyses en transformaties normaal uitgevoerd worden.

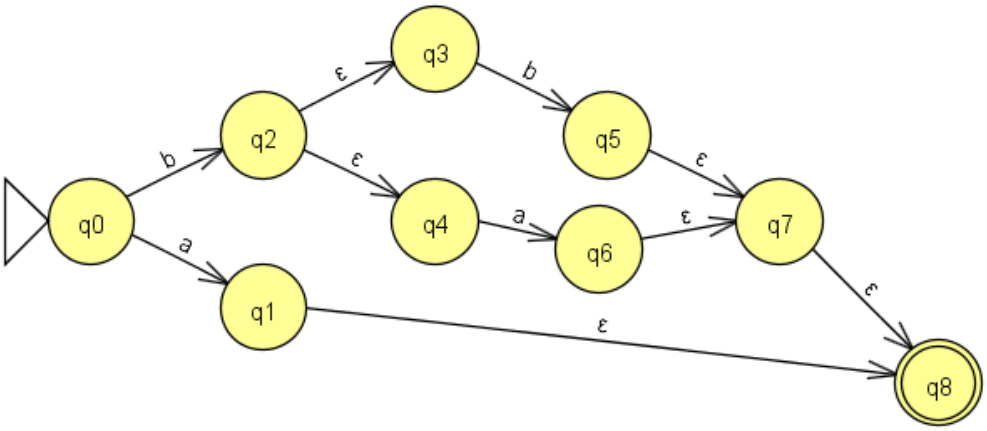
1.7 Vergelijkingen

Bij elke set vergelijkingen passen één of meerdere termen uit onderstaand lijstje. Welke?

1. voorwaarts
2. achterwaarts
3. cascading analysis
4. liveness analysis
5. available expression analysis
6. generational collection
7. worklist
8. reaching definition analysis
9. dominator analysis
10. initialisatie normaal gezien met lege sets
11. initialisatie normaal gezien met volle sets

vergelijkingen	bijpassende termen
$in[n] = \bigcup_{p \in pred[n]} out[p]$ $out[n] = gen[n] \cup (in[n] - kill[n])$	1,8,10
$in[n] = \bigcap_{p \in pred[n]} out[p]$ $out[n] = gen[n] \cup (in[n] - kill[n])$	1, 5, 11
$in[n] = use[n] \cup (out[n] - def[n])$ $out[n] = \bigcup_{s \in succ[n]} in[s]$	2, 4, 10
$D[n] = \{n\} \cup \left(\bigcap_{p \in pred[n]} D[p] \right)$	1, 9, 11
$in[n] = gen[n] \cup (out[n] - kill[n])$ $out[n] = \bigcup_{s \in succ[n]} in[s]$	

1.8 NFA



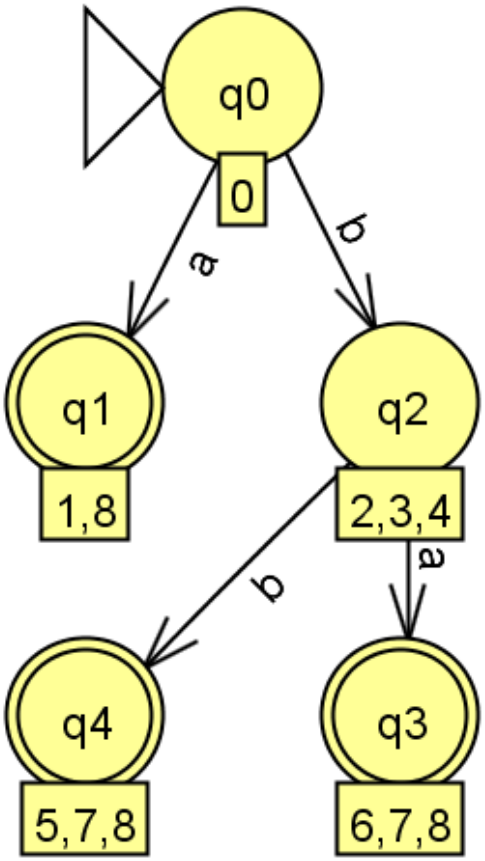
1. Converteer deze NFA naar een DFA.
2. Welke reguliere expressie wordt hiermee voorgesteld?

Antwoord

1. Gebruik dezelfde methodologie als in vraag 1.4.
2. De reguliere expressie die voorgesteld wordt is

```
a+ba+bb
```

ToDo: hoe bekomen



1.9 Parsing - I

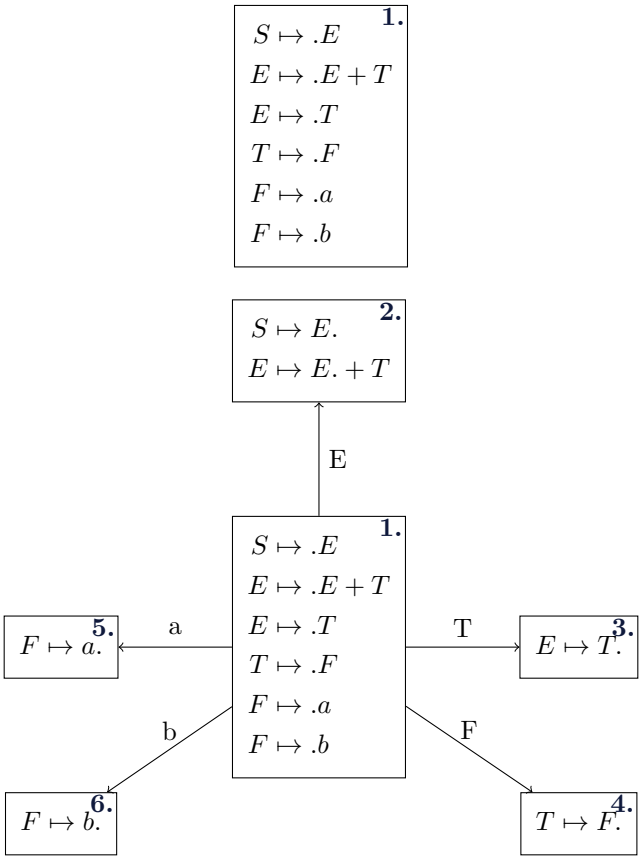
Gegeven de grammatica

$$
\begin{aligned}
S &\mapsto E \\
E &\mapsto E + T \\
E &\mapsto T \\
T &\mapsto F \\
F &\mapsto a \\
F &\mapsto b
\end{aligned}
$$

1. Welke zijn de items in de initiële toestand (1) van de LR(0) parser generator?
2. Teken de toestanden die bereikt worden na één shift of reduce actie van de LR(0) parser generator vanuit toestand 1. Teken ook de items in elke toestand.

Antwoord

1. Elke productieregel wordt een item in deze initiële toestand voor deze grammaatica.
2. Vanuit de eerste toestand zijn er enkel shift acties mogelijk.



1.10 Parsing - II

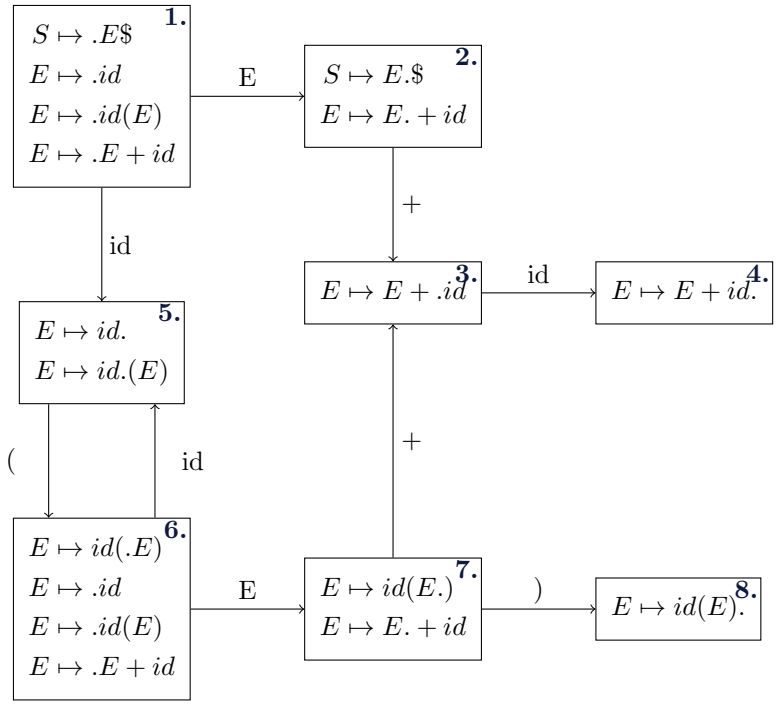
Gegeven de volgende context-vrije grammatica:

0. $S \mapsto E\$$
1. $E \mapsto id$
2. $E \mapsto id(E)$
3. $E \mapsto E + id$

1. Bouw een LR(0) DFA voor deze grammatica.
2. Is dit een LR(0) grammatica? Toon aan Waarom.
3. Is dit een SLR grammatica? Toon aan Waarom.
4. Is dit een LR(1) grammatica? Toon aan Waarom.

Antwoord

1. De LR(0) toestandenautomaat:
2. Dit is geen LR(0) grammatica. De LR(0) parsetabel (tabel 1.1) toont aan dat er een shift-reduce conflict is in toestand 5 voor symbool (.



	id	(	)	+	$	S	E
1	s5						g2
2				s3	a		
3	s4						
4	r3	r3	r3	r3	r3		
5	r1	s6, r1	r1	r1	r1		
6	s5						g7
7			s8	s3			
8	r2	r2	r2	r2	r2		

Tabel 1.1: De LR(0) parsetabel.

3. Voor de SLR parsetabel moet eerst de follow set bepaald worden van elke niet-terminaal dat niet S is.

   - FOLLOW(E) = +)

   De reductie voor elke niet-terminaal moet nu enkel uitgevoerd worden voor de terminalen in de follow set. De SLR parsetabel (tabel 1.2) toont aan dat er geen shift-reduce conflicten zijn. Dit is een SLR grammatica.

4. Als een grammatica SLR is, is het ook een LR(1) grammatica vanwege de hiërarchische ordening.

	id	(	)	+	$	S	E
1	s5						g2
2				s3	a		
3	s4						
4			r3	r3			
5		s6	r1	r1			
6	s5						g7
7			s8	s3			
8			r2	r2			

Tabel 1.2: De SLR parsetabel.

1.11 Liveness en registerallocatie

Gegeven volgend programma:

```
1. a = 1
2. b = 2
3. c = a + b
4. d = a
5. e = c + 2*b
6. f = b + e
7. return [d, f]
```

1. Doe de liveness analyse.
2. Teken de interferentiegraaf.
3. Maak een registerallocatie via graafkleuring, met het minimum aantal registers.

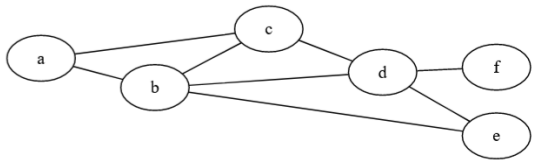
Antwoord

1. Deel het programma op in individuele instructies en bepaal per instructie zijn succesor set, use set en def set. Pas daarna het iteratief algoritme toe om de in en out sets te berekenen.

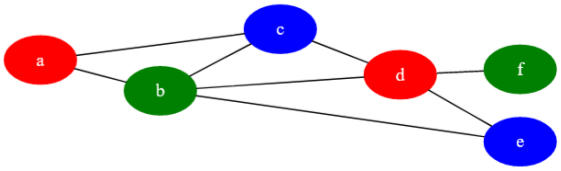
$$in[n] = use[n] \cup (out[n] - def[n])$$
$$out[n] = \bigcup_{s \in succ[n]} in[s]$$

	Succ	Use	Def	Out	In
7	/	d,f	/		d,f
6	7	b,e	f	d,f	b,d,e
5	6	b,c	e	b,d,e	c,b,d
4	5	a	d	c,b,d	a,b,c
3	4	a,b	c	a,b,c	a,b
2	3	/	b	a,b	a
1	2	/	a	a	/

2. De interferentiegraaf wordt bekomen door variabelen te verbinden die op hetzelfde moment live kunnen zijn (in set). De graaf is te zien op figuur 1.1



Figuur 1.1: De interferentiegraaf.



Figuur 1.2: De gekleurde interferentiegraaf.

3. Er zijn minimum 3 registers nodig aangezien er zeker drie variabelen op hetzelfde moment nodig zullen zijn (afleiden uit de in set). Volgende procedure wordt uitgevoerd:
   (a) Simplify **f**.
   (b) Simplify **e**.
   (c) Simplify **d**.
   (d) Simplify **c**.
   (e) Simplify **b**.
   (f) Simplify **a**.
   (g) Select **a** - rood.
   (h) Select **b** - groen.
   (i) Select **c** - blauw.
   (j) Select **d** - rood.
   (k) Select **e** - blauw.
   (l) Select **f** - groen.

   De resulterende graaf is te zien op figuur 1.2. Er kan dus afgeleidt worden dat c en d

1.12 Registerallocatie

1. Beschrijf kort de verschillende fasen van het registerallocatie-algoritme.
2. Wat is de betekenis van "moves"?
3. Hoe worden ze vermeden/weggewerkt?
4. Geef de 2 algoritmen + tekening.

Antwoord

1. Er zijn twee varianten van het registerallocatie-algoritme. Eerst wordt het algoritme uitgelegd zonder *conservative coalescing*, daarna met *conservative coalescing*. Beide algoritmen hebben een restrictie op het aantal kleuren (K) die gebruikt mogen worden.

   (a) Als er geen *conservative coalescing* gebruikt wordt, dan zijn er 5 fasen:

   i. **Build**: deze fase stelt de interferentiegraaf op door behulp van liveness analyse.

   ii. **Simplify**: Per iteratie in deze stap wordt er een knoop op een stack geplaatst en die minder dan K buren heeft. Die knoop zal zeker een kleur kunnen krijgen. Deze stap wordt herhaald totdat er ofwel geen knopen meer zijn in de graaf ofwel als er geen knopen meer zijn die minder dan K buren hebben.

   iii. **Spill**: Als alle resterende knopen K of meer buren hebben, dan wordt een willekeurige knoop gekozen om te *spillen* en wordt deze ook op de stack geplaatst. Spilling zorgt ervoor dat de temporary in het geheugen bewaard zal worden in plaats van het register, dus de interferentie tussen de temporary en alle andere temporaries verdwijnt.

   iv. **Select**: Vanaf dat elke knoop zich op de stack bevindt, worden ze één per één terug afgehaald en krijgen ze een kleur toegekend. Knopen die met de **simplify** opdracht op de stack geplaatst zijn kunnen zeker een kleur krijgen. Wanneer een knoop met de **spill** opdracht op de stack geplaatst wordt, is er geen garantie dat deze gekleurd kan worden. Als ze niet gekleurd kan worden praten we over een echte spill in plaats van een potentiële spill.

   v. **Start Over**: Als de **select** fase niet alle knopen heeft kunnen kleuren (door het plaatsvinden van echte spills), wordt het algoritme opnieuw uitgevoerd met een licht gewijzigd programma: de temporary die de spill veroorzaakte wordt nu in het geheugen opgeslagen na elke definitie en uit het geheugen gelezen voor elke use. Dit zorgt voor meer temporaries, maar ze hebben elk een kortere liveness range zodat ze minder interferen met andere temporaries.

   (b) Er kan ook *conservative coalescing* toegepast worden. Dit is het samenvoegen van twee move-gerelateerde knopen die het kleuren van de graaf zeker niet moeilijker zal maken. Het algoritme bevat nu twee extra stappen, alle andere stappen zijn hetzelfde als zonder *conservative coalescing*:

   i. **Build**

   ii. **Simplify**

   iii. **Coalesce:** Knopen die samen move-gerelateerd zijn kunnen samengevoegd worden. Er kunnen hier twee heuristieken gebruikt worden om knopen a en b samen te voegen:

   - heuristiek van Briggs: als de resulterende knoop ab minder dan K buren heeft met K of meer buren.
   - heuristiek van George: als elke buur t van a ofwel een buur is van b ofwel een knoop is met minder dan K buren.

iv. **Freeze:** Wanneer zowel **simplify** als **coalesce** niet meer van toepassing zijn, kunnen de move-gerelateerde knopen bevrozen worden. Dit komt er op neer dat de move-interferentie tussen elke andere knoop die buur is verwijdert wordt, zodat de graaf vereenvoudig is.

v. **Spill**

vi. **Select**

vii. **Start Over**

2. Een move komt voor wanneer een temporary naar een andere temporary gekopieerd wordt: $t \leftarrow z$.

3. Via het registerallocatie-algoritme met *conservative coalescing* kunnen zulke operaties weggewerkt worden. Inderdaad, knopen die *gecoalesced* zijn krijgen hetzelfde register toegewezen.

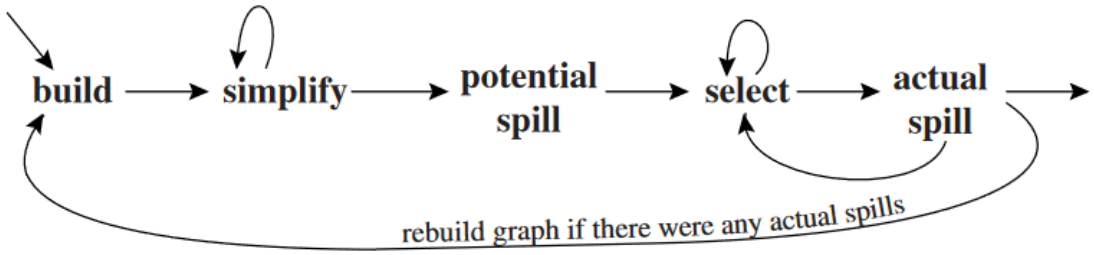
$$z \leftarrow x \oplus y$$
$$t \leftarrow z$$

In dit geval zullen z en t hetzelfde register krijgen (bv $r3$):

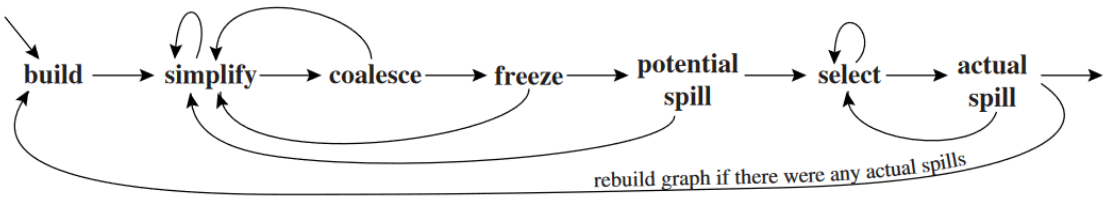
$$r3 \leftarrow x \oplus y$$
$$r3 \leftarrow r3$$

zodat de laatste toekenning geschrapt kan worden.

4. • Registerallocatie zonder *conservative coalescing*.



• Registerallocatie met *conservative coalescing*.



1.13 Bevriezen

In welke fase van de compiler wordt de term bevriezen (freeze) gebruikt? Wanneer wordt deze techniek toegepast, en wat is het nut ervan?

Antwoord

Bevriezen heeft betrekking tot registerallocatie. In de interferentiegraaf kunnen move-gerelateerde verbindingen zijn. Wanneer een knoop met een move-gerelateerde verbinding bevrozen wordt, worden alle move-gerelateerde verbindingen van die knoop verwijderd. Dit heeft als gevolg dat eventueel de move operatie twee verschillende registers nodig heeft. Het wordt gebruikt om de interferentiegraaf eenvoudiger te maken.

1.14 Bereikende definities

1. Wat zijn bereikende definities (reaching definitions)?
2. Wat zijn de gen- en kill-verzamelingen voor de statements voor volgende types van statements?

Statement s	gen[s]	kill[s]
$d : t \leftarrow b \oplus c$		
$d : t \leftarrow M[b]$		
$M[a] \leftarrow b$		
if a relop b goto L_1 else goto L_2		
goto L		
$L :$		
$f(a_1, ..., a_n)$		
$d : t \leftarrow f(a_1, ..., a_n)$		

3. Worden reaching definities gebruikt bij
   (a) eliminatie van gemeenschappelijke subexpressies?
   (b) copy propagatie?
   (c) dode code eliminatie?

   Geef alle mogelijkheden. Verklaar uw antwoord, zo mogelijk met een voorbeeld.
4. Geef de vergelijkingen van de in- en out-verzamelingen voor een statement n.
5. Gegeven twee opeenvolgende statements p en n in een basisblok. Bereken de vergelijkingen van de in- en out-verzamelingen van het blok pn van de tweede opeenvolgende statements.

Antwoord

1. Een reaching definition voor een statement n zijn alle statements s_i die een waarde toekennen aan een temporary, waarbij de waarde nog steeds beschikbaar is in n.
2. Stel $defs(t)$ alle statements waarin t gedefinieerd wordt:

Statement s	gen[s]	kill[s]
$d : t \leftarrow b \oplus c$	{d}	defs(t) - {d}
$d : t \leftarrow M[b]$	{d}	defs(t) - {d}
$M[a] \leftarrow b$	{}	{}
if a relop b goto L_1 else goto L_2	{}	{}
goto L	{}	{}
$L :$	{}	{}
$f(a_1, ..., a_n)$	{}	{}
$d : t \leftarrow f(a_1, ..., a_n)$	{d}	defs(t) - {d}

3. (a) Nee.
   Een reaching definition heeft enkel betrekking tot definities van temporaries. Om gemeenschappelijke subexpressies te elimineren moeten de reaching expressions bepaald worden. Een expressie $t \leftarrow x \oplus y$ (binnen een knoop s) bereikt een knoop n als er een pad bestaat van s naar n waarbij er in geen enkele knoop op dat pad een toekenning is aan x of y.

(b) Ja.

Een definitie $n : t \leftarrow t \oplus x$ en $d : t \leftarrow z$ kan vervangen worden door $n : t \leftarrow z \oplus x$ als er een pad bestaat van d naar n en als er geen andere definitie van t statement n bereikt.

(c) Nee.

Dode code elmination verwijdert statements die nergens gebruikt worden. Stel een definitie $s : a \leftarrow b$. Als a niet live-out s is, dan kan s verwijderd worden. Dit wordt geïmplementeerd via de out-verzamelingen bij liveness analyse.

4. Voor een statement n zijn de vergelijkingen van de in- en out-verzamelingen

$$in[n] = \bigcup_{p \in pred[n]} out[p]$$
$$out[n] = gen[n] \cup (in[n] - kill[n])$$

Hierbij is $pred[n]$ de verzameling statements die n als directe opvolger hebben.

5. Als p de enige voorganger is van n, dan kan eerst de vergelijking opnieuw geschreven worden:

$$\begin{aligned}
out[n] &= gen[n] \cup (in[n] - kill[n]) \\
in[n] = out[p] &= gen[p] \cup (in[p] - kill[p]) \\
out[n] &= gen[n] \cup ((gen[p] \cup (in[p] - kill[p])) - kill[n]) \\
out[n] &= gen[n] \cup (gen[p] - kill[n]) \cup (in[p] - (kill[p] \cup kill[n]))
\end{aligned}$$

Dus de vergelijkingen worden:

$$in[pn] = gen[p] \cup (in[p] - kill[p])$$
$$out[pn] = gen[pn] \cup (in[pn] - kill[pn])$$

1.15 Constantenpropagatie

Veronderstel een statement $d : t \leftarrow c$ met c een constante. Een ander statement n gebruikt t, zoals in $n : y \leftarrow t \oplus x$. Onder welke voorwaarden mag men statement n herschrijven als $y \leftarrow c \oplus x$?

1. Geef de precieze voorwaarden.
2. Definieer de gebruikte termen.

Antwoord

1. Statement d moet in eerste geval statement n bereiken. Verder mag er geen andere definitie van t statement n bereiken.
2. Om te bepalen of een andere definitie van t statement n bereikt, worden *reaching definitions* gebruikt. Dit is de verzameling van statements s_i, waarvan de waarde van de variabele die gedefinieerd wordt door dat statement, bereikt kan worden in n.