Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

R. van den Born

Struktuur behoudende data flow analyse op programma's met GOTO- statements

Afdeling Informatica Noot CS-N8401 Januari

Structure maintaining data flow analysis on programs with Goto statements

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

STRUKTUUR BEHOUDENDE DATA FLOW ANALYSE OP PROGRAMMA'S MET GOTO-STATEMENTS

REINIER VAN DEN BORN

Centrum voor Wiskunde en Informatica, Amsterdam

Om efficiënte (micro)code te genereren is het nodig gegevens te verkrijgen over parallellisme en data flow in een source-programma. Ook de struktuur van het programma moet bewaard blijven. De hier gepresenteerde graaf geeft een flexibel en overzichtelijk beeld van deze gegevens.
Ook wordt een nieuwe data flow analyse methode geïntroduceerd, waarmee de afbeelding geïmplementeerd kan worden. De methode wordt niet gehinderd door GOTO-statements van welke vorm dan ook.

1980 MATHEMATICS SUBJECT CLASSIFICATIE: 68B10. 69 B14, 69 D 44
1982 CR CATEGORIES: B.1.4., D.3.4.
TREFWOORDEN: high level data flow analyse, (micro) codegeneratie, optimalisatie.

Noot CS-N8401
Centrum voor Wiskunde en Informatica
Postbus 4079, 1009 AB Amsterdam

VOORWOORD

Dit rapport beschrijft de resultaten van een stage aan het Centrum voor Wiskunde en Informatica (CWI, voorheen Mathematisch Centrum). Behalve, dat het als rapport door het CWI uitgegeven wordt, dient het ook als doktoraalskriptie van ondergetekende. Daartoe is een aparte uitgave gemaakt, die bovendien voorzien is van een supplement. Dit supplement, "Implementatie van een Struktuur-Behoudende Data Flow Analyse op S_YALLL-Programma's", is dus niet in het CWI rapport opgenomen. Dit lijkt niet bezwaarlijk, omdat het alleen van belang is voor diegenen, die behoefte hebben aan een gedetailleerde uitwerking van het hier gepresenteerde algoritme, bijvoorbeeld om het te implementeren.

Het project, waaraan ik tijdens de stage gewerkt heb, stond onder leiding van Marleen Sint. Zij en Arthur Veen hebben mij gedurende de stage en het schrijven van dit rapport begeleid. Ik ben beiden veel dank verschuldigd voor de tijd en aandacht, die ze hieraan besteed hebben. Ook ben ik het CWI dankbaar voor de ter beschikking gestelde faciliteiten.

Reinier van den Born

INHOUD

1 INLEIDING

Dit rapport beschrijft een afbeelding van programma's, geschreven in de taal S_YALLL, op een graaf. De afbeelding is ontworpen als onderdeel van een microcodegenerator, waarbij de graaf dient als invoer voor de eigenlijke codegenerator (zie figuur 1). Hoewel de generatie van microcode specifieke eisen aan de graaf stelt, is hij ook bruikbaar voor andere doeleinden, waaronder andere codegeneratie systemen.

Naast de beschrijving van de afbeelding wordt ook een algoritme voor de afbeelding gepresenteerd. Onderdeel van het algoritme vormt een zogenaamde data-flow-analyse. Deze analyse wordt uit een ongebruikelijke hoek benaderd, waardoor enkele opmerkelijke resultaten geboekt worden. Ook het gebruik van deze nieuwe data-flow-analyse methode hoeft zich niet te beperken tot deze toepassing.

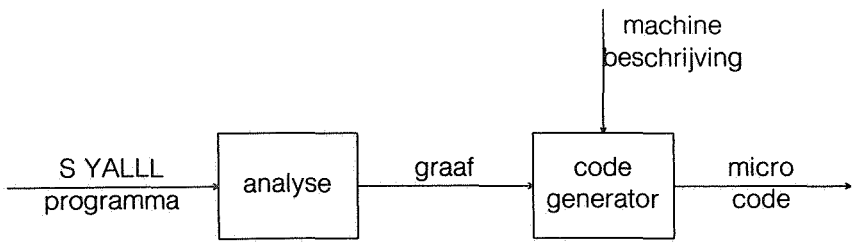
1.1 Microcodegeneratie

Het microprogramma vormt een laag tussen de machine-taal en de hardware van de meeste computers. De machine-instructies worden door het microprogramma geïnterpreteerd. Inefficiënties in deze interpretatie zijn van grote invloed op de snelheid van de machine. Een microcodegenerator is een programma, dat code in een zekere 'source'taal (in dit geval S_YALLL) vertaalt naar microcode voor een gegeven machine. Belangrijk daarbij is dus, dat de uiteindelijke code efficiënt is.

Een belangrijke factor bij microcodegeneratie is het laag niveau parallellisme, dat in microprogramma's aanwezig is. Microinstrukties besturen de hardware van een computer. Dat wil zeggen dat in een microinstructie aangegeven kan worden welke poorten er geopend of gesloten moeten zijn. Twee assignments, bijvoorbeeld, kunnen soms volbracht worden door één microinstruktie, die twee poorten tegelijk opent. Om efficiënte code te genereren moet een generator in staat zijn dit soort parallellisme in het source-programma te vinden en te coderen.

Een ander probleem, dat 'normale' compilers ook kennen, is de registerallocatie. Een extra moeilijkheid hierbij is de veelal asymmetrische hardware architectuur. Voor een goede registerallocatie is het noodzakelijk, dat er een voldoende hoeveelheid gegevens beschikbaar is over de gegevensstromen in het source-programma.

Het doel van het bovengenoemde project is een microcodegenerator, die niet aan een bepaalde machine gebonden is. Naast een source-programma wordt ook een beschrijving van de machine gegeven. In figuur 1 is dit schematisch weergegeven. (Het totale project is groter, maar de overige delen zijn niet interessant in dit verband.) Een gevolg van deze benadering is, dat de source-taal geheel machine-onafhankelijk moet zijn. Behalve, dat dit heeft geleid tot de keuze van de taal S_YALLL als source-taal, heeft dit tot gevolg, dat de registerallocatie niet aan de programmeur kan worden overgelaten (hetgeen in soortgelijke projecten wel eens gebeurt). Bovendien moeten alle machine afhankelijke truukjes, die door een microprogrammeur uitgehaald kunnen worden, nu door de generator bedacht worden.



Figuur 1 Het relevante gedeelte van de microcodegenerator.

1.2 De afbeelding

De afbeelding verschaft de codegenerator de gegevens om bovenstaande taken naar behoren uit te kunnen voeren. Besloten is, dat de volgende gegevens direct uit de graaf af te lezen moeten zijn:

- De oorspronkelijke struktuur van het programma. Dat wil zeggen, dat de nesting van statements en gegevens betreffende die statements behouden blijft.
- De mobiliteit van statements. Hoever kan een instruktie binnen het sourceprogramma verschoven worden zonder de semantiek van het programma aan te tasten?
- De levensduur van variabelen. Is een variabele op een gegeven plaats in het programma nog in leven? Met andere woorden: wordt de waarde, die de variabele daar heeft, later nog gebruikt?

In de volgende secties wordt bekeken waarvoor deze gegevens gebruikt kunnen worden en hoe ze uit het source-programma afgeleid kunnen worden.

1.2.1 De struktuur van het programma

Een bekende optimalisatie-techniek is het verwijderen van statements uit lussen in het programma. Als de programmeur gebruik gemaakt heeft van while-statements, dan zijn deze lussen gemakkelijk te vinden. Maar als de struktuur van het programma verloren is gegaan, dan is een analyse op de control flow graaf nodig om ze terug te vinden. Voor andere statements zijn vergelijkbare situaties te bedenken. Ook zal hieronder blijken, dat het behoud van de struktuur de bewegingsvrijheid van de statements vergroot.

De struktuur wordt gevonden door het source-programma te ontleden (parseren).

1.2.2 Het verschuiven van statements

De verschuifbaarheid van statements geeft een indruk van het parallellisme in het programma. Immers als het toegestaan is twee statements in willekeurige volgorde te executeren mogen ze ook tegelijk geëxecuteerd worden.

De mogelijkheid om statements te verplaatsen wordt door twee aspecten beperkt, namelijk door de control flow binnen een programma en door de zogenaamde 'data-afhankelijkheid' van statements. Deze aspecten worden nu apart besproken.

Control flow

Als zich ergens een splitsing of samenkomst van de control flow voordoet, dan mag een instructie niet zonder meer over dat punt heen geschoven worden. Immers dan worden opdrachten in sommige gevallen niet uitgevoerd, terwijl dat in het oorspronkelijke programma wel gebeurde. Met als gevolg, dat de betekenis van het programma gewijzigd wordt.

Wel mag een instructie soms over een tijdelijke splitsing heen geschoven worden. Dus van een punt voor de splitsing naar een punt na de bijbehorende samenkomst. Een voorbeeld hiervan is een gestruktureerd statement. Als daarin geen wilde sprongopdrachten voorkomen, dan kan een instructie over dat statement heen geschoven worden. Voorwaarde is wel, dat er geen data-afhankelijkheden optreden (zie hieronder).

Het aardige van deze soort tijdelijke splitsingen is, dat ze uit de syntax van de taal af te leiden zijn en er dus geen analyses op de vorm van de control flow graaf nodig zijn. Ook een argument om de struktuur van het programma te behouden, dus.

Data afhankelijkheid

Twee statements zijn data-afhankelijk, als beide een waarde aan dezelfde variabele toekennen (de variabele definiëren), of, als de één een variabele definieert en de ander die variabele gebruikt. In hoofdstuk 3 wordt een meer formele definitie gegeven.

Neem nu twee definities van dezelfde variabele. Als de volgorde verwisseld wordt, zal de

variabele na executie van beide statements een andere waarde hebben, dan wanneer de oorspronkelijke volgorde aangehouden zou zijn. Een later gebruik van de variabele geeft dan verkeerde resultaten. Dergelijke argumenten gelden ook als het om een toekennig en een gebruik gaat (in beide mogelijke volgorden). In het algemeen mogen definities van dezelfde variabele niet onderling van plaats verwisselen en mogen gebruiken van die variabelen niet relatief ten opzichte van die definities verplaatst worden. Kortom als twee statements data-afhankelijk zijn, dan mogen ze niet verwisseld worden.

1.2.3 Levensduur van variabelen

Voor een efficiënte registerallocatie is het noodzakelijk te weten op welk moment een eenmaal toegewezen register vrij komt voor een hernieuwde toewijzing. Dit moment komt overeen met het moment waarop de waarde van de variabele niet meer gebruikt zal worden in toekomstig uit te voeren instructies.

De levensduur van variabelen verandert zodra de statements verschoven worden. In de graaf wordt deze informatie echter op een dusdanige manier weergegeven, dat de gegevens geldig zijn, zolang bij het verschuiven bovenstaande beprekingen in acht genomen worden.

Voor het bepalen van de levensduur van variabelen is een data-flow-analyse van het programma nodig. Omdat dit het meest ingewikkeld is, zal een groot deel van het algoritme hier aan gewijd zijn.

1.3 Implementatie

Aan de implementatie van de afbeelding zijn weinig eisen gesteld, behalve natuurlijk, dat hij moet voldoen aan de specificatie van de afbeelding. Met name aan de snelheid waarmee de graaf gegenereerd wordt zijn geen voorwaarden gesteld. Het zal echter blijken, dat het algoritme in sommige gevallen weliswaar trager is dan andere methoden, maar dat de snelheid niet beïnvloed wordt door de vorm van de control flow van het programma. Dit in tegenstelling tot andere methoden, die vaak problemen ondervinden bij het behandelen van sprongopdrachten (GOTO-statements). Omdat in het omvattende project, de microcode-generator, voor de programmeertaal SUMMER als implementatietaal gekozen is, ligt het voor de hand de afbeelding ook in die taal te implementeren.

De ontwikkeling van het algoritme heeft meer tijd gekost dan voorzien en daarom is het niet tot een implementatie gekomen. Wel is een gedetailleerd ontwerp gemaakt, dat beschreven wordt in [Born84].

1.4 Indeling van de beschrijving

De beschrijving van de afbeelding wordt aan de hand van de source-taal S YALLL gegeven. (Dat wil natuurlijk niet zeggen, dat de methode niet voor andere talen te gebruiken is.) Daarom wordt in hoofdstuk 2 een korte beschrijving van S YALLL gegeven. Hierbij zal alleen aandacht besteed worden aan die aspecten van de taal, die voor de afbeelding van belang zijn.

In hoofdstuk 3 worden enige termen gedefinieerd, die in de resterende hoofdstukken gebruikt zullen worden.

Hoofdstuk 4 bevat een functionele specificatie van de afbeelding. Dat wil zeggen, dat het verband tussen de source-tekst en de graaf vastgelegd wordt. Ook zal aangegeven worden hoe de code-generator de informatie, die hij nodig heeft, aan de graaf kan onttrekken.

Hoofdstuk 5 geeft een indruk van het algoritme, dat de graaf opbouwt. De nadruk zal daarbij liggen op de data-flow-analyse, die nodig is om een gedeelte van de graaf te maken. Andere aspecten, die betrekking hebben op de meer specifieke gebieden van de afbeelding, worden slechts kort aangestipt.

Tenslotte worden de graaf en het algoritme in hoofdstuk 6 geëvalueerd en vergeleken met bestaande methoden. Met name wordt ook de snelheid van de analyse-methode bekeken.

2 S_YALLL

S_YALLL is een assembler-achtige taal uitgebreid met voorgedefinieerde types en enige gestruktureerde statements. De taal is afgeleid van de door Patterson, Lew en Tuck in [PaLT76] gepresenteerde taal YALLL. Omdat YALLL niet helemaal machine-onafhankelijk is, zijn er enige veranderingen in aangebracht. Tegelijk is de taal op enkele punten uitgebreid, onder andere met een case-statement. Het resultaat is de taal S_YALLL (de S staat voor 'source'). De taal is volledig beschreven in [Sint].

In dit hoofdstuk zullen alleen die aspecten van S_YALLL aan bod komen, die voor de bouw van de graaf van belang zijn. Zo wordt dus bijvoorbeeld niet ingegaan op de verschillende data-typen, die de taal kent. De volledige syntax van S_YALLL is te vinden in de appendix.

2.1 Statements

S_YALLL-statements laten zich in drie groepen onderverdelen: de primitieven, de jumps en de gestruktureerde statements. Elk statement kan voorafgegaan worden door een label.

Primitieven

De eerste groep bestaat uit primitieve, gegevens manipulerende statements. Net als in assembler bestaat zo'n statement uit de naam van het statement gevolgd door een aantal operanden. In dit verband komt alleen de functie van de operanden aan de orde. Bij de bespreking wordt de volgorde aangehouden waarin de operanden in de primitieven optreden (zie syntax in appendix).

destination — De variabele waar het resultaat van de operatie aan toegekend wordt. Behalve bij de zogenaamde externals (zie hieronder) is er altijd een destination. Het is de eerste operand.

source — Een variabele, wier waarde gebruikt wordt in de operatie. Behalve externals heeft iedere primitieve één of twee sources.

count — (alleen bij shifts) Variabele, waarvan de waarde de lengte van de shift bepaalt.

flags — (optioneel) Vlaggen, die afhankelijk van het resultaat van de operatie gezet moeten worden. De keus is beperkt tot een aantal voorgedefinieerde vlaggen.

In deze groep zijn de externals buitenbeentjes. Externals zijn de instructies, die met het geheugen of een extern medium (afhankelijk van implementatie) communiceren. Hun syntax is enigzins afwijkend van de anderen. Afhankelijk van de richting van de communicatie moet of een destination of een source gegeven worden. Bij geheugenreferenties is bovendien nog een adres nodig.

Enige voorbeelden:

```
ADD a, b, c       ; a ← b + c
ADD a, b, c <Z>   ; idem en zet Z-vlag bij som 0
SRL a, b, c       ; a ← b, c posities naar rechts verschoven
LOAD a, b         ; a ← inhoud geheugen op adres b.
```

Jumps

Jumps houden zich bezig met de control flow. Hieronder vallen achtereenvolgens

`JUMP` label — de onconditionele sprongopdracht. Er zijn geen beperkingen, behalve, dat het niet toegestaan is een statement (bijv. een IF) in te springen of van de ene tak naar de andere (bijv. van THEN naar ELSE gedeelte) te springen.

`CALL` label

`RTN` De subroutine call en return. Het label bij de CALL moet op programma niveau staan. Het mag dus niet binnen een statement staan.
Een RTN mag overal staan. Wordt een RTN aangetroffen zonder dat daar een CALL aan te pas gekomen is, dan wordt gereageerd als of er een EXIT staat (zie hieronder).

`EXIT` De halt opdracht. Programmaexecutie wordt onmiddellijk gestopt. Mag overal geplaatst worden.

Gestruktureerde statements

Tenslotte zijn er nog de gestruktureerde statements. Dit zijn de statements, die statements kunnen bevatten, waardoor de nesting ontstaat. In S_YALLL zijn er drie voorhanden:

`IF` Het IF-statement bestaat minimaal uit een test. Het kan uitgebreid worden met ELIF's en eventueel een ELSE gedeelte. Een test bestaat uit een eenvoudige vergelijking van twee operanden.

`CASE` Het CASE-statement bestaat uit een 'selector' en minstens een tak. Een tak bestaat uit minstens een 'key' gevolgd door nul of meer statements. Het statement kan afgesloten worden met een DEFAULT tak. Er is geen 'fallthrough' van een tak naar volgende takken, zoals in de programmeertaal C. Een key moet een constante zijn.

`WHILE` Het WHILE-statement bestaat uit een test (dezelfde als bij het IF-statement) gevolgd door de loop body. Er is geen break-achtig statement voorhanden, maar dat kan gesimuleerd worden door waar gewenst uit de loop te springen met een JUMP naar het volgend statement.

2.2 Programmastruktuur

Een S_YALLL programma bestaat uit een rij declaraties, gevolgd door een of meer (gelabelde) blokken. De declaraties betreffen de globale variabelen en constanten. Deze mogen in het hele programma gebruikt worden.

Ieder blok bestaat uit een rij lokale declaraties en een rij statements. De hier gedeclareerde variabelen en constanten mogen alleen binnen het blok gebruikt worden. Het is toegestaan globale variabelen lokaal opnieuw te declareren.

Lokale variabelen hebben een eigenschap, die ze onderscheidt van de lokale variabelen uit hogere programmeertalen. Bij het verlaten van het blok, waarin ze gedeclareerd zijn, worden ze allemaal ongedefinieerd. Daarbij is het niet van belang of dat via het laatste statement of door middel van een JUMP of CALL gebeurt. Hetzelfde vindt plaats, als er vanuit het blok naar het label van het blok gesprongen wordt.

Er bestaat dus een verschil tussen een sprong vanuit een blok naar het label van het blok en een sprong vanuit een blok naar het eerste statement van het blok. Het tweede overleven de lokale variabelen, het eerste niet.

De reden voor deze ongewone behandeling van lokale variabelen is, dat het bijhouden van een stack voor deze variabelen veel tijd (en ook ruimte) kost, hetgeen niet acceptabel gevonden werd. Door deze behandeling van lokale variabelen wordt het schrijven van recursieve procedures aanzienlijk bemoeilijkt. Als een programmeur recursieve procedures wil schrijven, dan zal hij zelf een stack bij moeten houden van de te behouden lokale variabelen.

De blokken kunnen gebruikt worden om net gestruktureerde programma's te schrijven. Daartoe wordt het hoofdprogramma in het eerste blok gezet, afgesloten met een EXIT, en elke subroutine in een volgend blok, afgesloten met een RTN. Er is echter niets in S_YALLL, dat de programmeur verbiedt de meest wanstaltige programma's te schrijven.

2.3 Een voorbeeld van een S_YALLL-programma

In deze sectie wordt een voorbeeld gegeven van een programma in S_YALLL. Dit voorbeeld zal in de volgende hoofdstukken zo mogelijk ook gebruikt worden. Het programma kan niet "net" genoemd worden, maar dat is de bedoeling ook niet. Want nette programma's geven veel minder problemen en zijn dus niet zo illustratief.

Als voorbeeld van de merkwaardige mogelijkheden binnen S_YALLL kan opgemerkt worden, dat de subroutine-aanroep in het tweede blok (CALL COMP_OTHERS) vervangen zou kunnen worden door een sprong (JUMP COMP_OTHERS). Omdat de CALL direct gevolgd wordt door de RTN van het GOOD-blok, verandert de betekenis van het programma daardoor niet. De RTN's van het derde blok zouden dan bij beide subroutines COMP_OTHERS en GOOD horen.

De rest van deze sectie is gewijd aan het algoritme, dat aan het voorbeeld ten grondslag ligt. Dit is bedoeld voor degenen, die willen weten, wat er zich in het programma afspeelt. Om een indruk van S_YALLL te krijgen, volstaat een blik op de source-tekst.

Het voorbeeld speelt de passieve kant van het spelletje 'Mastermind'. Dat houdt in, dat het programma een door de speler gegeven rij kleuren met een rij kleuren in het geheugen vergelijkt. Er worden twee scores afgeleverd: het aantal keren, dat op dezelfde plaats dezelfde kleur staat, en het aantal keren, dat dezelfde kleuren op verschillende plaatsen staan. Kleuren mogen vaker dan één keer in een rij voorkomen.

Om de score te bepalen worden de kleuren uit de ene rij als het ware gekoppeld aan eenzelfde kleur in de andere rij. Een kleur mag maar aan één ander gekoppeld zijn. Een mogelijke koppeling van twee kleuren op dezelfde plaats heeft voorrang.

Het algoritme ziet er als volgt uit:

0) Initialisatie (p1, p2, p3)
1) Nog niet alle kleuren van de speler gehad (t1),
   dan naar 2),
   anders naar 8)
2) Neem volgende kleur van speler en kleur op dezelfde plaats uit geheugen (p4, p6).
   Als deze afgezien van een eventueel merk hetzelfde zijn (p5, t2),
   dan naar 3),
   anders naar 6)
3) (GOOD) Verhoog 'good' (p10).
   Als de kleur in het geheugen niet gemerkt is (t3),
   dan naar 4),
   anders naar 5)
4) Merk kleur (p12, p13).
   Ga naar 1)
5) Verlaag 'misplaced' (een 'misplaatste' koppeling is vervangen door een 'goede').
   Naar 6).
6) (COMP_OTHERS) Vergelijk de kleur van de speler met alle ongemerkte kleuren in het geheugen.
   Wordt dezelfde kleur gevonden (t5),
   dan naar 7),
   anders naar 1)
7) (MISPLACED) Verhoog misplaatst. Merk kleur. (p17-p19)
   Naar 1)
8) Druk resultaat af (p8, p9).

In het onderstaande programma wordt een kleur voorgesteld door 4 bits, zodat er 16 mogelijke kleuren zijn. Iedere kleur heeft bovendien een vijfde bit om aan te geven of de kleur al gekoppeld is. Het vijfde bit van de kleuren van de speler is altijd 0, dus als alle bits vergeleken worden, dan komen in praktijk alleen de ongekoppelde kleuren uit het geheugen in aanmerking (vgl t5).

```
                    CONST  first=1, last=4
                    INTEGER good, misplaced, address          (v1,v2,v3)
                    BITS[5:1] guess, color, unflagged         (v4,v5,v6)
SCORE:              BEGIN
                        MOVE good, 0                                (p1)
                        MOVE misplaced, 0                           (p2)
                        MOVE address, first                         (p3)
                        WHILE address <= last                     (w1,t1)
                            LOAD color, address                     (p4)
                            AND  unflagged, color, ?01111           (p5)
                            READ guess                              (p6)
                            IF guess = unflagged                    (i1,t2)
                                CALL GOOD                           (j1)
                            ELSE
                                CALL COMP_OTHERS                    (j2)
                            ENDIF
                            ADD address, address, 1                 (p7)
                        ENDWHILE
                        WRITE good                                  (p8)
                        WRITE misplaced                             (p9)
                        EXIT                                        (j3)
                    END

GOOD:               BEGIN
                        ADD good, good, 1                          (p10)
                        IF color[5] = ?1                          (i2,t3)
                            SUB misplaced, misplaced, 1            (p11)
                            CALL COMP_OTHERS                        (j4)
                        ELSE
                            OR color, color, ?10000                (p12)
                            STORE color, address                   (p13)
                        ENDIF
                        RTN    (j5)
                    END

COMP_OTHERS:        BEGIN
                        INTEGER address                             (v7)
                        BITS[5:1] color                             (v8)
                        MOVE address, first                        (p14)
                        WHILE address <= last                     (w2,t4)
                            LOAD color, address                    (p15)
                            IF guess = color                       (i3,t5)
                                JUMP MISPLACED                      (j6)
                            ENDIF
                            ADD address, address, 1                (p16)
                        ENDWHILE
                        RTN    (j7)
MISPLACED:              ADD misplaced, misplaced, 1                (p17)
                        OR color, color, ?10000                    (p18)
                        STORE color, address                       (p19)
                        RTN    (j8)
                    END
```

3 DEFINITIES

In dit hoofdstuk worden enige begrippen gedefinieerd, die gebruikt zullen worden bij de beschrijving van de afbeelding en het algoritme. Deze definities zullen gegeven worden in termen van (de syntax van) S_YALLL.

Soms wordt een term in een definitie gevolgd door een engelse tussen haakjes. Deze laatste wordt in de SUMMER-source van [Born84] voor dat begrip gebruikt.

3.1 De control flow graaf

Om te beginnen wordt de executievolgorde van de statements in het programma vastgelegd in de control flow graaf. Deze graaf zal in het algoritme niet gebruikt worden en zal dus niet 'echt' gemaakt worden. Hij dient meer om een intuïtief begrip te formaliseren. Dankzij deze graaf wordt menige definitie sterk vereenvoudigd.

In de control flow graaf worden alle <test>s, <primitive>s en <jump>s uit het source-programma gerepresenteerd door een knoop. Een gestruktureerd statement uit het programma wordt geassocieerd met de verzameling van alle knopen, die een <test>, <primitive> of <jump> uit het statement representeren.

Een case-statement zullen we in deze opvatten als een if-statement. Daartoe moet een speciale test geintroduceerd worden.

OF <key1>: ..: <keyN>:

wordt gezien als:

(EL)IF <selector> ∈ { <key1>, .., <keyN>}

en het DEFAULT-gedeelte wordt het ELSE-gedeelte van het nieuwe if-statement.

De kanten in de control flow graaf geven de executievolgorde weer. Daarop zijn twee uitzonderingen gemaakt. De CALL krijgt niet de subroutine-entry als opvolger, maar wordt als een soort <primitive> behandeld, en de RTN krijgt in deze graaf helemaal geen opvolger.

Alvorens over te gaan tot de bepaling van de opvolgers even een hulpdefinitie:

Def. De *eerstvolgende kandidaat* na een gegeven punt in de source-tekst is
- als op dat punt een ELIF of ELSE volgt: de eerstvolgende kandidaat na de bij de ELIF of ELSE horende ENDIF.
- als op dat punt een ENDWHILE volgt: de <test> van het while-statement.
- anders: de eerstvolgende (in de tekst) <test>, <primitive> of <jump>.

Merk op, dat in het eerste geval de definitie recursief is. Het kan zijn, dat een eerstvolgende kandidaat niet bestaat.

De regels voor het opvolgerschap zien er als volgt uit:

- RTN's en EXIT's hebben geen opvolger.
- Een <primitive> of CALL heeft één opvolger en wel de eerstvolgende kandidaat.
- Een JUMP heeft als enige opvolger de eerstvolgende kandidaat na het label, dat bij de JUMP gespecificeerd wordt.
- Een <test> van een statement heeft twee opvolgers, namelijk:
  1) de eerstvolgende kandidaat na de <test>
  2) de eerstvolgende kandidaat na de volgende bij statement behorende ELIF of ELSE of, als deze er niet zijn, na de bij het statement behorende ENDIF of ENDWHILE.

Als een kandidaat niet bestaat, is er geen opvolger.

In deze graaf wordt ook door middel van knopen vastgelegd, waar zich de 'ingang' van een programma, subroutine of statement bevindt. Deze speciale knopen worden entry's genoemd. Entry's hebben geen voorganger.

Def. De *entry* van het programma is een knoop met als opvolger de eerstvolgende kandidaat na de BEGIN van het eerste <block>.

Def. De *entry* van een subroutine is een knoop met als opvolger de eerstvolgende kandidaat na het label van de subroutine.

Def. De *entry* van een gestruktureerd statement is een knoop met als opvolger de eerste <test> van het statement.

3.2 Body's, takken en segmenten

Body's

Programma's, gestruktureerde statements en subroutineaanroepen kunnen op twee manieren bekeken worden. 'Van buiten af', als een eenheid, die net als een primitieve een of andere aktie tot gevolg heeft, of 'aan de binnenkant', als zijnde samengesteld uit meerdere statements en eventueel testen.

In het vervolg zullen we de woorden programma en statement reserveren voor de buitenkant en 'body' toevoegen, als we over de inhoud spreken en dan met name de bovenste nestingslaag daarvan.

Programma- en statement-body's komen overeen met met nestingsniveaus. De programma-body bestaat uit de statements, die op het hoogste nestingsniveau staan. Dus

Def. De *body* van het programma bestaat uit alle <statement>s, die niet in een ander statement staan.

Een statement-body bestaat uit de test(en) van het statement en alle statements, die precies één nestingsniveau lager liggen, dan het statement zelf.

Def. De *body* van een statement B bestaat uit alle statements en testen A, die aan de volgende voorwaarden voldoen:
- A staat in B
- ieder ander statement omvat òf zowel A als B, òf geen van beide.

Op een soortgelijke manier kunnen we de body van een subroutine definiëren. Een probleem is echter, dat deze body niet zozeer door de syntax van S_YALLL bepaald wordt, als wel door de semantiek. Het betreft immers die statements, die door een subroutineaanroep geaktiveerd kunnen worden. Om ze te vinden moeten dus de control flow paden vanaf de entry gevolgd worden.

Ook in deze body wordt natuurlijk een zo hoog mogelijk nestingsniveau geëist.

Def. Een statement maakt deel uit van een *subroutine-body* , als alle <test>s, <primitive>s en <jump>s uit het statement te bereiken zijn langs een pad, dat in de subroutine-entry begint. Bovendien mag het statement geen deel uitmaken van de body van een statement met dezelfde eigenschap.

Doordat de body niet door de syntax wordt bepaald, korrepondeert de body van een subroutine niet met een nestingsniveau. Wel kan eenvoudig ingezien worden, dat een subroutine altijd een deelverzameling is van de programma-body. Immers een subroutine begint op het programma-niveau en als een deel van een statement deel uitmaakt van een subroutine-body, dan maakt het hele statement deel uit van de body.

Ter illustratie een aantal body's uit het voorbeeld programma:

programma:		p1, p2, p3, w1, p8, p9, p10, i2, etc.
statement:	w1:	t1, p4, p5, p6, i1, p7
	i2:	t3, p11, j4, p12, p13
subroutine:	GOOD:	p10, i2, j5
	COMP_OTHERS:	p14, w2, j7, p17, p18, p19, j8

Takken

Statement-body's bestaan dus uit testen en statements. In if en case statements kunnen de statements verdeeld worden in (in de tekst) aaneengesloten stukken. Deze stukken zullen takken worden genoemd.

Def. Een *tak* (branch) van een if- of case-statement bestaat uit alle statements uit de body, die in de tekst tussen een <test> of ELSE en de volgende ELIF, ELSE of ENDIF staan.

Een tak vormt dus de productie van een <labeled_statement>* van het if of case-statement. Takken zijn disjunct.

Een tak begint na een test of ELSE. Aan iedere test kan dus een tak toegewezen worden, waarna er wellicht nog een ELSE-tak overblijft. Van dit verband zal later soms gebruik gemaakt worden, als er gesproken wordt van de tak van een test of iets dergelijks.

Een soortgelijke verdeling van een body van een while statement levert slechts een tak op. Deze rij statements wordt dikwijls de while-body genoemd. In het vervolg zullen wij echter spreken van de 'loop' en het begrip body reserveren voor de loop gecombineerd met de test.

Def. De *loop* van een while statement bestaat uit alle statements, die in de body staan.

Segmenten

De rij statements (en tests) van een programma- of statement-body kan zo in stukken gehakt worden, dat de control flow in zo'n stuk lineair is. Daarbij is de interne control flow van de statements niet van belang. Het enige criterium is de onderlinge control flow. Een dergelijk stuk van een body wordt een segment genoemd.

Het segment is een belangrijk begrip. Om de lezer een goed idee te geven van hetgeen het inhoudt zullen we er wat langer bij stil blijven staan. Om te beginnen een wat informele definitie, die berust op een intuïtief begrip van executievolgorde. Aan het einde van deze paragraaf zal het segment formeel gedefinieerd worden.

> Een *segment* bestaat uit een geordende rij statements en/of tests uit een programma of statement-body zodanig, dat een element uit die rij geexecuteerd wordt, d.e.s.d.a. de executie van zijn voorganger uit de rij net beëindigd is.

Hieruit kunnen de volgende conclusies getrokken worden:

1) Alleen het eerste element mag meer dan één voorganger hebben en alleen het laatste meer dan één opvolger. Een gevolg hiervan is, dat een test altijd alleen in een segment komt te staan. Een test heeft immers twee opvolgers en de voorganger is òf een test òf een statement buiten de body. Deze zogenaamde test-segmenten zullen we in de volgende punten buiten beschouwing laten.
2) Executie mag niet midden in een segment kunnen blijven steken. Ieder statement moet geëxecuteerd worden na het eerste. In de praktijk betekent dit, dat alleen het laatste statement EXIT's en RTN's mag bevatten.
3) Uit het woordje 'net' moet de conclusie getrokken worden, dat er tussen de executie van twee opeenvolgende statements uit een segment geen ander statement geëxecuteerd mag worden. De enige 'zijsprongen', die toegestaan zijn, zijn die door middel van CALLs. Voorwaarde is dan wel, dat terugkeer uit de subroutine gegarandeerd is: de subroutine mag niet tot een EXIT leiden. Een

gevolg van bovenstaande is, dat een segment bepaald wordt door het eerste en het laatste statement.

4) De grootte van een segment ligt niet vast. Een triviale indeling in segmenten is die, waarin ieder segment uit één statement bestaat. Iedere aaneengesloten deelrij van een segment is weer een segment. Twee segmenten vormen samen een segment als het eerste statement van het tweede segment altijd direct na het laatste statement van het eerste segment geëxecuteerd wordt.

In de hoop, dat nu een redelijk idee is gevormd van wat een segment is, wordt nu overgegaan tot een formele definitie in termen van de control flow graaf. Daarvoor eerst een hulpdefinitie.

Def. Een subroutine *bevat een EXIT*, als er een control flow pad van de subroutine-entry bestaat naar een EXIT of naar een CALL van een EXIT-bevattende subroutine.

Def. Een *segment* is een niet lege deelverzameling van een programma- of statement-body, waarvan de elementen in een zodanige volgorde gezet kunnen worden, dat voor elk paar opeenvolgende elementen A en B geldt:
a) iedere knoop in A heeft een opvolger (een test twee) en iedere opvolger ligt in A of in B
b) iedere voorganger van een knoop in B ligt in A of in B
c) A bevat geen CALLs naar EXIT-bevattende subroutines.
d) B is niet gelabeld met de naam van een subroutine.

De eis, dat een knoop in A een opvolger moet hebben, verbiedt de aanwezigheid van EXITs en RTNs in alle statements behalve het laatste.

Voorbeelden van de verdeling van body's in segmenten kunnen gevonden worden in de hoofdstukken 4 en 5

3.3 Definities en gebruiken

Statements gebruiken en definiëren variabelen. Een statement dat een variabele gebruikt of definieert, zullen we ook respectivelijk een gebruik (use) en een definitie (definition) noemen.

Primitieven

Als eerste klasse van statements, worden de <primitive>s besproken.

Def. Een <primitive> *gebruikt* een variabele x, als
- x als <source>, <count> of <address> gebruikt wordt.
- x de speciale variabele 'memory' is en de <primitive> een 'LOAD' of 'STORE' is.
- x de speciale variabele 'extern' is en de <primitive> een 'READ' of 'WRITE' opdracht is.

Een <primitive>,die een variabele gebruikt, zal soms ter onderscheid een primitief gebruik genoemd worden.

Def. Een <primitive> *definieert* een variabele x, als
- x als <destination> gebruikt wordt of x onder de <flags> voorkomt
- x de speciale variabele 'memory' is en de <primitive> een 'STORE' opdracht is
- x de speciale variabele 'extern' is en de <primitive> een 'READ' of 'WRITE' opdracht is.

Net als bij het gebruik zullen we soms van een primitieve definitie spreken.

Def. Een <primitive>, die een variabele x niet definieert, heet *transparant* (voor x).

Tests

Def. Een <test> *gebruikt* een variabele x, als x een <test_operand> is of de <selector>

Def. Een <test> is *transparant* voor alle variabelen

Zichtbaar

Def. Een gebruik heet *zichtbaar* (exposed) vanaf een gegeven knoop, als er een pad bestaat van de knoop naar het gebruik waar slechts transparante knopen liggen.

Met andere woorden als er geen primitieve definities tussen liggen.

CALLs

Een CALL willen we graag zien als een normaal <primitive>-achtig statement in plaats van een statement, dat de control flow beinvloedt. Vandaar, dat gebruik en definitie ook voor CALLs gedefinieerd wordt.

Def. Een CALL *gebruikt* een variabele x, als er een gebruik van x zichtbaar is vanaf de entry van de subroutine.

Deze definitie is recursief. Het gebruik mag ook een CALL zijn.

Def. Een CALL *definieert* een variabele x, als er een pad bestaat van de entry van de subroutine naar een definitie van x

Def. Een CALL of subroutine heet *transparant* voor een variabele x, als een pad door de subroutine (= van entry naar een RTN) bestaat zodanig, dat alle knopen op dat pad transparant zijn voor x.

Gestruktureerde statements

Behalve <primitive>s en CALLs kunnen gestruktureerde statements (if, case en while) ook als een gebruik of definitie gezien worden. Dit wordt op bijna dezelfde manier bepaald, als bij de CALLs.

Def. Een gestruktureerd statement *gebruikt* een variabele x als een gebruik van x, dat in het statement staat, zichtbaar is vanaf de entry van het statement.

Def. Een gestruktureerd statement *definieert* een variabele x als zich in het statement een definitie van x bevindt.

Def. Een gestruktureerd statement heet *transparant* voor een variabele x, als er een pad door het statement bestaat zodanig, dat alle knopen op het pad transparant zijn voor x.

3.4 Data-afhankelijkheid

Def. Twee statements uit een segment noemen we *data-afhankelijk* als een van beide een variabele definieert, die de ander definieert of gebruikt. Bovendien mag er geen definitie van die variabele tussen beide statements staan.

Opgemerkt moet worden, dat er in de definitie geen volgorde tussen beide statements aangegeven wordt. Als we wel op de volgorde letten, kunnen we dus drie vormen van data-afhankelijkheid onderscheiden, namelijk:

def-def beide statements definiëren dezelfde variabele.

use-def een gebruik van een variabele gevolgd door een definitie van die variabele.

def-use een definitie gevolgd door een gebruik.

4 DE AFBEELDING

In dit hoofdstuk wordt een functionele specificatie van de afbeelding gegeven. Dat wil zeggen, dat gegeven een S_YALLL-programma bepaald wordt hoe de bijbehorende graaf er uit komt te zien. Met name de secties 4.1, 4.2 en 4.5 houden zich hiermee bezig.

Ter illustratie van deze secties is in 4.6 het programma uit hoofdstuk 2 op een graaf afgebeeld. Het bekijken van dit voorbeeld tijdens het lezen van bovengenoemde secties kan deze secties wat begrijpelijker maken.

Aan het einde van het hoofdstuk zullen we met de specificatie in de hand laten zien, hoe de codegenerator de gevraagde informatie uit de graaf kan afleiden.

De in het vorig hoofdstuk gedefinieerde begrippen hebben betrekking op de source-tekst. Hier wordt de source-tekst op een graaf afgebeeld. Eigenschappen, die op het origineel gedefinieerd zijn, zullen ook aan het betreffende beeld worden toegekend. Zo zal bijvoorbeeld van een gebruik gesproken worden, als een knoop bedoeld wordt waarop een gebruik is afgebeeld.

4.1 Knopen en segmenten

In deze sectie wordt beschreven, hoe de graaf er uitziet zonder kanten. Dit om een idee te geven van de struktuur van de graaf, de nesting enzovoorts, zonder verwarring te scheppen met de kanten.

De struktuur van de graaf laat zich door de volgende regels beschrijven:

- Ieder statement en test wordt in de graaf afgebeeld op een knoop. Een test van een case-statement wordt gerepresenteerd door een rij knopen met per knoop een test van de vorm <selector> = <key>.
- De knopen die samen een body van een statement of het programma vormen, zijn verdeeld in segmenten (volgens in de volgende sectie geformuleerde regels).
- Ieder segment heeft bovendien één IN-knoop en één OUT-knoop. Deze knopen representeren de plaatsen waar het segment resp. 'betreden' en 'verlaten' kan worden. In nader te bepalen gevallen is het laatste statement van een segment in de OUT-knoop ondergebracht.
- De segmenten, die samen de body van een statement vormen, staan in de knoop, die het statement representeert.
- Er is een speciale knoop, die het programma representeert. In deze knoop staan de segmenten, die de programma-body vormen.

Op het hoogste nestingsniveau is er dus één knoop, de programma-knoop. Daar staat een rij segmenten in, elk opgebouwd uit een IN-knoop, een OUT-knoop en een of meer test- of statement-knopen.

De knopen, die een <test>, <primitive> of <jump> representeren bevatten slechts gegevens over hun origineel. De knopen waarop de gestruktureerde statements afgebeeld zijn bevatten bovendien de segmenten, waarin hun body is onderverdeeld. Deze segmenten zijn weer opgebouwd uit een IN-, een OUT- en een of meer statement- of test-knopen.

Op iedere nestingslaag uit de source-tekst is dus een segmentlaag aangebracht, maar afgezien daarvan wordt de nesting precies zo in de graaf gerepresenteerd, als die in de source-tekst gevonden werd.

4.2 Segmentindeling

In deze sectie wordt nader bepaald hoe een body in segmenten wordt verdeeld. Ook worden de voorwaarden, waaronder een knoop in de OUT-knoop terecht komt, vastgelegd.

Een body wordt als volgt in segmenten verdeeld.

- Iedere test staat in een apart segment. De test-knopen van een case statement staan per tak samen in één segment.
- Iedere tak, loop of programma-body wordt in stukken van opeenvolgende (in de tekst) statements verdeeld. 'Grenzen' worden op de volgende plaatsen gelegd:
  - Voor een statement dat voorafgegaan wordt door een label. Het label wordt aan het segment gehangen.
  - Na een RTN-, EXIT- of JUMP-statement.
  - Na een CALL van een subroutine, die een EXIT bevat (zie definitie in sectie 3.2.-segmenten).
  - Na een statement, dat een RTN, EXIT, CALL als boven of JUMP naar een label buiten het statement bevat.

Indien een van de laatste drie voorwaarden opgaat, dan wordt het statement in de OUT-knoop gezet.

4.3 Entry's en exit's

Zoals gezegd zijn de IN- en OUT-knoop de plaatsen, waar de segmenten 'betreden' en 'verlaten' worden. Als er een gestruktureerd statement in de OUT-knoop staat is deze plaatsbepaling niet precies genoeg. De plaats waar een segment verlaten kan worden wordt een exit genoemd. Analoog wordt de toegang tot een segment de entry genoemd.

Def. De *entry* van een segment is zijn IN-knoop.

Def. De verzameling *exits* van een segment bestaat uit:
de OUT-knoop van het segment, als deze leeg is of een <jump> bevat.
of anders de exits van het gestruktureerde statement (zie hieronder), dat in de OUT-knoop staat.

Def. De verzameling *exits* van een tak, loop of programma-body bestaat uit exits van de segmenten, waaruit het is samengesteld, en wel:
de exits, waarin een JUMP naar een label buiten de tak of loop staat,
en de exits, waarin een EXIT of RTN staat,
en de lege exits van het laatste segment van de tak, loop of programma-body.

Def. De *exits* van een if- of case-statement zijn:
de exits van test-segmenten, die geen bijbehorende tak hebben of niet gevolgd worden door een ELIF of ELSE.
en de exits van de takken

Def. De *exits* van een while-statement zijn:
de exit van het test-segment
en de niet-lege exits van de loop.

Opmerkingen

a) Een exit is dus altijd een OUT-knoop, waarin zich óf niets óf een <jump> bevindt.

b) Een exit kan dus exit van meerdere segmenten op opeenvolgende niveaus zijn, maar is dan altijd in een OUT knoop van die niveaus te vinden. Met andere woorden, het is dan een OUT-knoop in een OUT-knoop .. in een OUT-knoop.

c) Als een exit een JUMP bevat, dan is het een exit op alle niveaus daarboven tot en met het niveau, waar het label staat. Het label kan niet op een lager niveau staan dan de JUMP, omdat we <statement>s niet binnen mogen springen.

d) De exits van een statement bepalen mede de segmentindeling. Heeft een statement een exit, die niet leeg is, dan bevat het statement een RTN, een EXIT, een EXIT-bevattende CALL of een JUMP naar buiten het statement. Dan wordt er dus een segmentgrens achter gelegd en komt het statement in de OUT-knoop terecht.

4.4 Control flow

De control flow in een segment is lineair, dus om de control flow in een body te beschrijven hoeven er alleen verbanden tussen de exits en entry's van de segmenten gelegd te worden. (Vgl. de gebruikelijke control flow grafen, waarin de knopen basic block's voorstellen [Kenn81]) Hiertoe wordt de hieronder gedefinieerde successor/predecessor relatie gebruikt.

Omdat deze relatie dient om de control flow tussen segmenten weer te geven, is alleen voor exits van segmenten, die niet tevens exit van de omvattende body zijn, een opvolger gedefinieerd. Dit zijn de exits, die een JUMP bevatten, en de lege exits van alle segmenten behalve het laatste uit de betreffende loop, tak of body. De opvolger van deze exits is de entry van een segment. Dit segment staat op het hoogste niveau, waarop de exit nog exit is. Wat preciezer:

- De successor van een exit, waarin een JUMP L staat, is de entry van het segment met dat label.
- De successor van een lege exit van een segment, dat niet het laatste is uit de tak, loop of programma body is de entry van het volgende segment.
- De successor van een lege exit van een loop is de entry van het test-segment van het while-statement.
- Als de bijbehorende tak of loop bestaat, dan is de eerste successor van de exit van een test-segment de entry van het eerste segment van die tak of loop.
- Als een test gevolgd wordt door een ELIF (ELSE), dan is de entry van het volgende test-segment (het eerste segment van de ELSE-tak) de andere successor van de exit van het test-segment. Anders is het een (lege) exit van het statement.

Opmerkingen

a) Als een exit geen exit van het omvattend statement is, dan heeft die exit een successor binnen dat statement.

b) Als een exit wel exit van het omvattend statement is, dan wordt de successor bepaald door de situatie op het hogere niveau.

c) Een exit waarin een EXIT of RTN staat krijgt geen opvolger.

d) Een exit van een statement, dat niet in een OUT-knoop staat heeft ook geen opvolger (een dergelijke exit moet leeg zijn, zie ook opmerking d) in de vorige sectie).

4.5 Kanten

We maken onderscheid tussen twee soorten kanten: data-afhankelijkheids-(d-a-) en data-flow-(d-f-)kanten. De eerste geven de data-afhankelijkheden binnen de segmenten weer, de tweede geven aan hoe de gegevens door de graaf stromen door middel van kanten in de segmenten. Alle kanten zijn gelabeld met de naam van een variabele.

Tussen twee knopen kan dus alleen een kant liggen, als ze in hetzelfde segment staan. Een body wordt gerepresenteerd door een geordende rij segmenten, waartussen geen kanten liggen.

Data-afhankelijkheid

Data-afhankelijkheidskanten worden tussen twee knopen A en B uit één segment gelegd als:

de statements, die door A en B gerepresenteerd worden, data-afhankelijk zijn.

en/of als:

A een CALL bevat of is en
B een gebruik of definitie van een variabele x is en
er geen gebruik van x zichtbaar is vanaf een CALL van dezelfde subroutine.

In sectie 4.7. wordt duidelijk waarom dit tweede criterium ook een data-afhankelijkheid aangeeft.

Data-flow

De plaatsing van data-flow-kanten is een minder triviale aangelegenheid. Eerst zullen we aangeven wat we onder gegevensstromen verstaan.

Er is een gegevensstroom tussen twee <primitive>s, indien de eerste een variabele definieert, die de tweede gebruikt, en er bovendien een pad in de control flow graaf van de eerste naar de tweede bestaat waarop slechts transparante knopen liggen. Met andere woorden, het programma kan zo geexecuteerd worden, dat de variabele, die bij de definitie een waarde krijgt, niet opnieuw gedefinieerd is op het moment dat het gebruik bereikt wordt.

Met de data-flow kanten worden nu de gegevenstromen in het programma als het ware in de graaf uitgetekend. De data-fow-paden zullen echter niet alle knopen aandoen, die in de control flow graaf op het pad liggen. Veelal worden deze knopen op een hoger nestingsniveau gepasseerd.

Er ligt een d-f-kant met label x van een knoop A naar een knoop B, beide in die volgorde deel uitmakend van hetzelfde segment, als er geen definitie van x tussen beide staat en aan een van de volgende voorwaarden wordt voldaan:

1) A definieert x of is de IN-knoop en B gebruikt x
2) A definieert x of is de IN-knoop, B is de OUT-knoop en
   - of (a) B bevat een gestruktureerd statement en er ligt een d-f-kant met label x aan de entry van het eerste test-segment van het statement.
   - of (b) B is een exit met een successor en aan de successor van B ligt een d-f-kant met label x
   - of (c) B is een exit zonder successor en aan het omvattend statement ligt een d-f-kant met label x

Volgens bovenstaande regels liggen er aan een OUT-knoop, waarin een RTN of EXIT staat, geen kanten.

Behalve deze kanten worden er extra kanten in subroutine-body's gelegd. Daarbij worden bijna dezelfde voorwaarden gehanteerd, als hierboven, alleen:

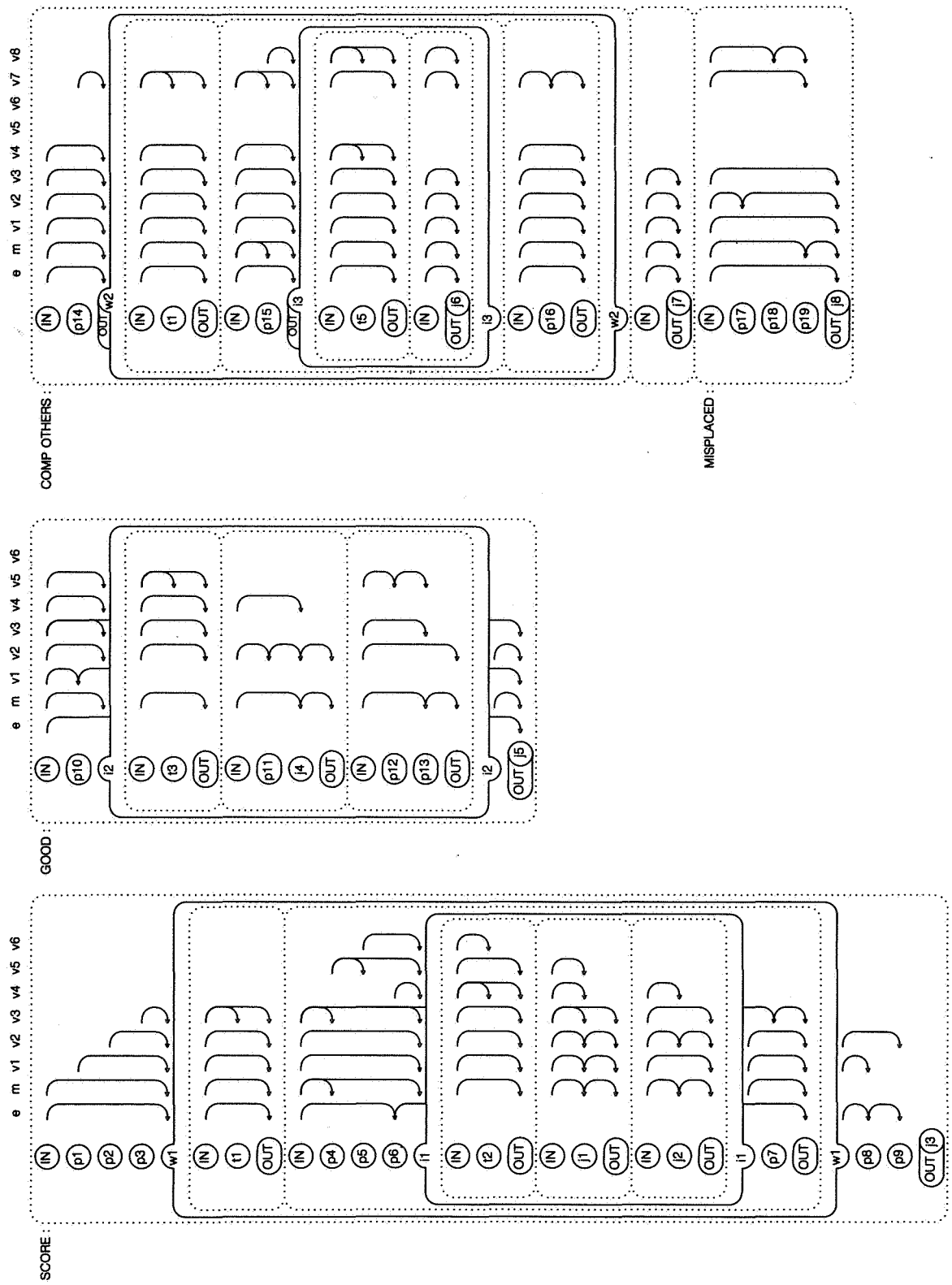
- Het segment, waarin A en B staan, moet deel uitmaken van de body van de betreffende subroutine.
- Aan 2) wordt toegevoegd:
  B is een exit, waarin een RTN staat, en
  er is een gebruik van x zichtbaar vanaf een CALL van de subroutine.

4.6 Het voorbeeld

Ter illustratie van het voorgaande is de graaf getekend voor het voorbeeld programma uit sectie 2.3. In figuur 2 staat een afbeelding van de graaf. Omdat de graaf nogal vol met kanten staat, is, in een poging tot het scheppen van orde, gekozen voor een nogal ongebruikelijke representatie. Deze wordt nu eerst toegelicht.

Stippellijnen zijn gebruikt om segmenten aan te geven, ononderbroken lijnen voor knopen en daarmee voor de nesting. Voor dat laatste zijn de knopen voor gestruktureerde statements op geblazen tot een bijna rechthoekige vorm en de inhoud van het statement is in die rechthoek geplaatst. De knopen bevatten een verwijzing naar het source programma. Zoals daar al aangegeven, geeft de eerste letter van deze verwijzing het type van de knoop aan. Als een knoop in een OUT-knoop staat is de laatste als het ware opengeknipt en tegen de eerste aangeplakt.

De pijlen in de figuur stellen data-flow-kanten voor. D-a-kanten zijn niet afgebeeld, omdat ze weinig interessant zijn, veelal samenvallen mat d-f-kanten en overvloedig in aantal zijn. Het aantal d-f-kanten is ook groot, vandaar dat ze gesorteerd zijn op label (variabele). Een kolom bevat alle kanten voor een zekere variabele. Bovenaan de kolom wordt aangegeven om welke variabele het gaat. (In plaats van de naam is een code gebruikt, die in het source-programma achter de declaraties terug te vinden is.) Op deze manier is eenvoudig te zien hoe de data-flow-paden van de diverse variabelen



Figuur 2. De graaf (een kwartslag gedraaid).

door de graaf lopen.

Bij de opgeblazen knopen eindigen inkomende kanten aan de bovenkant en beginnen uitgaande kanten aan de onderkant (vgl "v1"kanten in eerste segment naar en van de while-knoop). Een kant, die aan een dergelijke knoop voorbij gaat, stopt bij het bereiken van het contour van de betreffende knoop en begint weer aan de onderkant (vgl. "e"kant in het "GOOD"segment).

Knopen kunnen meerdere opvolgers hebben voor dezelfde variabele. Dat wordt in de figuur weergegeven door een pijl naar de verste opvolger en aftakkingen van die pijl bij de tussenliggende opvolgers. Dus in het eerste segment van w1 is er een "v3"kant van de IN- naar de t1- èn van de IN- naar de OUT-knoop. Dergelijke aftakkingen kunnen natuurlijk ook bij knopen met inhoud voorkomen (zie bijv. de "v3"kanten in het "GOOD"segment).

Tenslotte kan een knoop zowel een inkomende als een uitgaande kant hebben. Bij de kleine knopen vallen de uiteinden van de pijlen samen. Er is echter wel een duidelijke 'inkeping' te zien ter onderscheid van aftakkingen. In het "GOOD"segment loopt er dus een "v1"kant van de IN- naar de p10- en van de p10- naar de OUT-knoop.

4.7 Codegeneratie op grond van de graaf

Verwisselen

Twee statements mogen verwisseld worden, als ze deel uitmaken van hetzelfde segment en als er geen kant tussen ligt. Bovendien mag een statement, dat in de OUT-knoop staat, met geen enkel statement uit het segment verwisseld worden.

Hoe meer statements de codegenerator mag verwisselen, des te efficienter zal de gegenereerde code zijn. Het is dus zaak de segmenten zo groot mogelijk te maken. De indeling, die hierboven gegeven wordt, is redelijk efficient. Betere verdelingen zijn voorstelbaar. Op dit onderwerp wordt in hoofdstuk 6 nader ingegaan. Ook zal daar een methode besproken worden om de data-afhankelijkheid te reduceren.

Zoals gezegd vallen d-f- en d-a-kanten veelal samen. Het zou mooi zijn als de d-a-kanten niet nodig waren. Nu kan, na de bovengenoemde reductie van de data-afhankelijkheid, overwogen worden de d-a-kanten weg te laten. Alleen sommige use-def-afhankelijkheden worden dan niet in de graaf weergegeven, maar deze kunnen eenvoudig afgeleid worden van de aanwezige d-f-kanten. Als het uitvoeren van deze afleidingen tijdens de code-generatie niet als bezwaarlijk wordt gezien, kan hiervoor gekozen worden.

Merk op, dat het gebruik van JUMP's, RTN's en EXIT's de bewegelijkheid van statements reduceert. Een spaarzaam gebruik van die statements wordt dus beloond met grotere segmenten en daardoor (hopelijk) met efficientere code.

Levensduur

De levensduur van variabelen wordt gegeven door de d-f-kanten.

Als van een knoop in een segment een d-f-kant met label x uitgaat, dan is de variabele x vanaf die knoop in leven tot de knoop, die het uiteinde van de kant vormt. Een kant van de IN- naar de OUT-knoop duidt er dus op, dat de variabele het segment moet overleven.

Als een CALL zich tussen een definitie en een gebruik bevindt, dan moet die variabele in de subroutine body blijven leven. Daartoe zijn in de body's de speciale d-f-kanten aangelegd. (vgl. de 'extern'kanten in de subroutines in het voorbeeld) Hierbij is echter uitgegaan van de positie van de CALL in het source programma. Als een CALL verschoven wordt, zouden variabelen voortijdig het loodje kunnen leggen. Om dit te voorkomen wordt de verschuifbaarheid van de CALL aan banden gelegd door middel van extra d-a-kanten (zie 4.5: data-afhankelijkheids-kanten)

Dat deze kanten terecht d-a-kanten genoemd worden, kan als volgt verklaard worden. Als een variabele sterft door een subroutine-aanroep, dan krijgt hij de waarde 'ongedefinieerd'. In feite wordt

de variabele dus door de CALL gedefinieerd. Als de CALL als definitie van die variabele wordt gezien, dan volgen daaruit data-afhankelijkheden, die precies door de extra kanten worden aangegeven.

5 OVERZICHT VAN HET ALGORITME

In dit hoofdstuk wordt een globaal overzicht van het algoritme gegeven. Er zal vooral ingegaan worden op de samenhang tussen de deelalgoritmen. Niet alleen stelt dat de lezer in staat het algoritme beter te begrijpen, maar ook kunnen we ons daardoor in de beschrijving van het implementatie ontwerp in [Born84] voornamelijk bezighouden met een gedetailleerde beschrijving van wat er gebeurt in die diverse onderdelen.

Het programma moet ten eerste de geneste graaf opbouwen. Daarin moeten de d-a- en d-f-kanten worden aangebracht. Om dit te kunnen doen is het ook nodig, dat de control flow in de graaf wordt weergegeven. Voor dit alles worden vier fasen gebruikt. Deze worden in de secties 5.2 t/m 5.5 besproken. Daarvoor, in sectie 5.1, zal tot in een redelijk groot detail ingegaan worden op de manier waarop d-f-kanten gelegd zullen worden. In sectie 5.6 tenslotte zullen enkele speciale maatregelen aan deze methode toegevoegd worden.

In dit hoofdstuk worden de volgende eisen aan het aangeboden source-programma gesteld:

a) Het programma is syntactisch correct.
b) Alle identifiers (variabelen en constanten) worden gedeclareerd en gebruikt, zoals voorgeschreven is.
c) Ook het gebruik van de labels voldoet aan de gestelde normen. (Dus alleen sprongen naar minstens hogere niveaus, alle subroutine-entry's op het programma-niveau)
d) Er zijn geen onbereikbare statements (dode code).
e) Iedere subroutine heeft minstens één RTN, die bereikbaar is vanaf de entry van de subroutine. Dat wil niet zeggen, dat er geen EXIT's in de body mogen staan, maar wel, dat er altijd een mogelijkheid moet zijn om uit de subroutine-body naar een CALL terug te keren.

In hoofdstuk 6 zal kort besproken worden hoe het algoritme tegen dit soort fouten beschermd kan worden. Het algoritme kan hierbij een aktieve rol spelen door deze fouten op te sporen en (sommige) te corrigeren.

5.1 Data-flow-paden

De aanleg van data-flow-kanten begint bij een gebruik. Teruglopend langs control flow paden worden de kanten gelegd. Daar waar meerdere paden bij elkaar komen (in de looprichting een splitsing), worden alle richtingen één voor één onderzocht. Omdat de d-f-kanten samen, afgezien van segment- en nesting-overgangen, paden tussen definities en gebruiken vormen, zal in het vervolg ook van data-flow-paden gesproken worden.

Het leggen van kanten in een bepaalde richting houdt op, zodra zich een van de volgende gevallen voordoet:

- er wordt een niet-transparante definitie gevonden.
- er is geen mogelijkheid verder te gaan (bijv. de programma-entry is bereikt).
- op de plaats, waar een kant gelegd moet worden, wordt een kant met hetzelfde label aangetroffen.

Er zijn twee manieren waarop de laatste situatie kan ontstaan. Ten eerste kan zich een (recursie-)lus in het programma bevinden, waardoor het d-f-pad in aanleg zichzelf tegenkomt. Ten tweede kan de kant gelegd zijn doordat een ander gebruik (een deel van) de bijbehorende definities gemeen heeft.

In beide gevallen mag aangenomen worden, dat ook de kanten achter de gevonden kant al bestaan of op de nominatie staan om gelegd te worden. Het voorkomt dus, dat overbodige (dubbele) kanten gelegd worden, en garandeert bovendien, dat het algoritme in programma's met lussen termineert.

De aanleg van paden wordt soms gebruikt om eigenschappen van diverse objecten te bepalen. Zo gebruikt een gestruktureerd statement of CALL een variabele, als vanaf een gebruik in de body een pad aangelegd kan worden, dat langs de entry van de body loopt. Iets dergelijks geldt ook voor

het transparant zijn van statements en CALLs.

Een speciaal geval is het bereiken van de programma-entry. Behalve voor de speciale variabelen 'extern' en 'memory', duidt dat op de mogelijkheid, dat een variabele onvoldoende geinitialiseerd wordt. In dergelijke gevallen kan een boodschap gegenereerd worden. De programmeur kan dan nagaan of hij een fout heeft gemaakt of dat het zo bedoeld is.

5.1.1 Een pad in een segment

Het leggen van d-f-kanten in een segment vereist, dat de control flow in een segment is aangegeven. Per definitie is een segment geordend, maar deze ordening wordt in de graaf niet weergegeven. In plaats daarvan wordt een partiele ordening aangebracht.

Deze partiele ordening wordt weergegeven in de vorm van definitieketens. Deze ketens geven de omgekeerde control flow volgorde van definities van een bepaalde variabele aan. De ketens beginnen in de OUT-knoop, lopen terug langs de definities naar uiteindelijk de IN-knoop. Als een segment geen definities van een bepaalde variabele bevat, dan ligt er een kant van de OUT- naar de IN-knoop. Definitie-ketens worden tijdens de eerste fase aangelegd (zie 5.2)

Stel, dat vanaf een gegeven knoop uit de definitie-keten een pad gelegd moet worden. (Die knoop is dus een definitie of een OUT-knoop. Hoe het allereerste begin van het pad, bij het gebruik, gemaakt wordt, wordt in sectie 5.2 beschreven) Dan worden afhankelijk van het type van de knoop de volgende akties ondernomen:

De knoop is de OUT-knoop.

a) Als er al een kant ligt van de volgende knoop uit de definitieketen naar deze knoop, dan wordt het lopen gestaakt.
b) Anders wordt die kant gelegd en wordt de keten vanaf die volgende knoop verder afgelopen.

De knoop is een primitieve definitie.

Het pad is klaar.

De knoop representeert een if-, case- of while-statement

Eerst wordt vanaf iedere exit in de body van het statement een pad gelegd (zie sectie 5.1.2).
Dan wordt bekeken of het statement transparant is.
Zo ja, dan wordt verder gehandeld als bij de OUT-knoop.
Zo nee, dan is het pad klaar.

De knoop is de IN-knoop.

Het einde van de keten is bereikt, klaar dus.

De knoop kan geen jump zijn, omdat alle jumps (dus ook CALL's) ten tijde van het aanleggen van paden in een OUT-knoop staan. Zie de voorlopige segmentindeling in 5.2.1.

5.1.2 Paden in een statement- of programma-body

Om een pad door een body te leggen moet van segment op segment overgestapt worden. Om dit te kunnen doen is de successor/predecessor-relatie tussen de entry's en exit's in de graaf weergegeven met kanten. Deze control-flow-kanten worden tijdens de eerste fase aangelegd (zie 5.2).

Een CALL staat als gevolg van de voorlopige segmentindeling in een OUT-knoop. Een CALL staat dus in een exit. Maar hij kan ook zelf gezien worden als een soort exit. Vanaf de CALL maakt de control flow immers een omweg via de subroutine-body. Dit verband wordt ook in de graaf aangegeven. Ter onderscheid zal de CALL in zijn functie van exit met kleine letters geschreven worden, dus: call.

Het aanleggen van een pad in een body begint bij een exit of call. Aangenomen wordt dat de body helemaal afgebouwd is. Voor de definitie-ketens in de segmenten van de body betekent dat, dat ze beginnen in de OUT-knoop. Daar worden de volgende akties in de gegeven volgorde ondernomen:

1) Als in een exit, die een CALL bevat, gestart wordt, dan wordt eerst een pad in de bijbehorende subroutine-body gelegd (zie volgende sectie). Als de subroutine-body niet transparant is, dan is het pad klaar.

2) Als de exit of call op een lager niveau staat dan de body, dan wordt eerst een pad in de body van het statement in de OUT-knoop (één niveau lager dus) aangelegd. Natuurlijk wordt daarbij weer in dezelfde exit of call gestart.
(N.B. Dit betekent, dat de nesting recursief wordt afgedaald, tot op het niveau waarop de exit staat.)
Als die body niet transparant is, dan wordt het paden leggen verder gestaakt.

3) Het pad wordt lokaal (binnen het huidige segment) vanaf de OUT-knoop voortgezet zoals beschreven is in 5.1.1. Als daarbij de IN-knoop niet bereikt wordt, dan is het pad klaar

4) Het pad wordt vanaf alle predecessors van de segment-entry voortgezet. Als het segment gelabeld is met de naam van een subroutine, dan wordt het pad bovendien vanaf elke bijbehorende call verder aangelegd.

5.1.3 Paden in een subroutine-body

Als tijdens de aanleg van een pad een CALL aangetroffen wordt, dan wordt een omweg gemaakt langs de body van de subroutine. Dit niet alleen als de subroutine de variabele definieert (zoals bij een normaal statement) maar ook als er geen definitie te vinden valt. Dan immers moet aangegeven worden, dat de variabele de subroutine moet overleven (zie ook 4.7).

Eigenlijk worden er meerdere omwegen gemaakt. Elke omweg begint bij een RTN van de subroutine. Het aanleggen van paden vanaf zo'n RTN, de subroutine-body in, onderscheid zich op het volgende punt van de aanleg in een programma- of statement- body:

Alleen die predecessors, die in een segment staan, dat deel uit maakt van de subroutine body, komen in aanmerking voor verder zoeken. CALL's worden hierbij ook niet meegeteld.

5.1.4 Een voorbeeld

Om het bovenstaande te illustreren zal nu een klein voorbeeld behandeld worden. In de volgende sectie worden ook enkele voorbeelden gegeven, die een beeld van de boven geschetste methode geven.

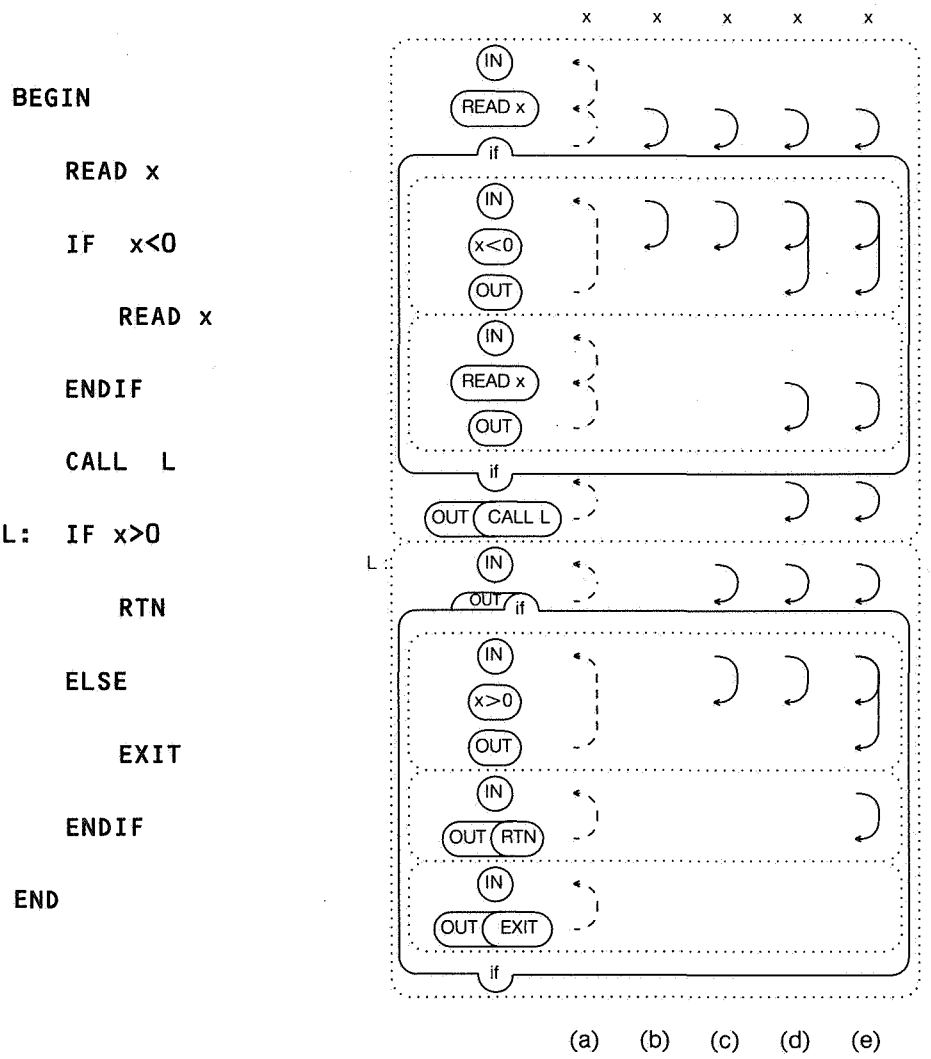
In figuur 3 zijn verschillende stadia van de aanleg van de paden naast elkaar in één graaf getekend. De stadia zijn de volgende:

a) De definitie-ketens voor x. Ter onderscheid zijn ze gestreept. Control flow kanten zijn niet getekend, omdat ze nogal voor de hand liggen en eerder verwarring dan duidelijkheid zouden geven. Merk wel op, dat er twee manieren zijn om bij L terecht te komen. Eerst via de CALL, en dan na de RTN, door de zogenaamde "fall-through".

b) Hier is het eerste pad getekend. Er is een kant van de test naar de IN-knoop gelegd. Het omvattend if-statement gebruikt 'x' dus ook. Dus wordt er een kant van het if-statement naar de volgende knoop uit de keten gelegd. Dat is een primitieve definitie, dus het pad is klaar.

c) De paden bij het tweede gebruik kosten wat meer inspanning. Het eerste gedeelte is analoog aan het bovenstaande, behalve, dat er nu geen definitie gevonden wordt en het pad dus voorlopig bij de IN-knoop van het L segment eindigt. Er zijn twee kandidaten om het pad verder te vervolgen. Dat zijn de exit en de call van het eerste segment.

d) De call wordt het eerst genomen. De call staat op het huidige niveau, dus wordt meteen de definitie-keten gevolgd. Het if statement is de eerste definitie. Dit statement heeft twee exits. De tweede leidt tot een definitie, de eerste (die van het test-segment) niet. Het if-statement is dus transparant. Het pad hoeft echter niet verder uitgebreid te worden, omdat er al een pad ligt.



Figuur 3 De aanleg van een pad

e) Nu wordt de exit van het eerste segment genomen. Daarin staat een CALL, dus moet het pad in de subroutine L gelegd worden, te beginnen bij de RTN's. Er is er maar één. Deze staat op een lager niveau, dus moet er een niveau afgedaald worden. Het omvattende if-statement blijkt transparant, dus moet op het oude niveau de definitie-keten gevolgd worden. Daar ligt al een pad, dus klaar. De subroutine is transparant, dus moet er vanaf de call verder gelegd worden. Ook daar worden kanten aangetroffen, zodat nu het werk er op zit.

5.2 De eerste fase: Ontleding en opbouw

Tijdens de eerste fase wordt het geraamte van de graaf gemaakt. Dat wil zeggen, dat de knopen, segmenten en body's worden gecreëerd. Daarbij wordt een voorlopige segmentindeling gehanteerd, die in 5.2.1 nader bekeken wordt.

Tegelijk worden in deze graaf definitieketens en control-flow-kanten aangebracht (zie 5.2.2). Verder wordt het merendeel van de d-a-kanten gelegd (5.2.3) en tenslotte wordt een gedeelte van de d-f-paden gemaakt (5.2.4).

5.2.1 Voorlopige segmentindeling

CALL's kunnen worden gezien als gestruktureerde statements, waarvan de body elders in het programma staat. Hierdoor kan, in tegenstelling tot andere statements, bij het aantreffen van een CALL-statement in het programma niet direkt bepaald worden, welke variabelen het definieert of gebruikt, en of het EXIT's bevat. Er kan dus niet, zoals dat bij normale statements gebeurt, direkt bepaald worden of de knoop volgens de segmentindeling van 4.2 voor een segmentgrens zorgt. Ook is onbekend in welke definitie-ketens de CALL opgenomen zou moeten worden.

Om deze reden wordt in de eerste fase iedere CALL in een OUT-knoop gezet. De CALL hoeft dan ook niet in definitieketens opgenomen te worden. Ook wordt het daardoor mogelijk tijdens de eerste fase al wat paden te leggen (In hoofdstuk 6 wordt kort ingegaan op de mogelijkheid géén kanten in de eerste fase te leggen).

Hierdoor wordt wel een nieuw probleem gecreëerd, namelijk het weghalen van segment-grenzen, maar wordt uitgesteld tot de laatste, geheel aan die taak gewijde, fase (zie 5.5).

Lokale variabelen mogen niet buiten hun block bestaan. Om te voorkomen, dat hiermee vergissingen ontstaan, is het handig als een block in de graaf gerepresenteerd wordt door een geheel aantal segmenten. Hierdoor hoeft in ieder geval niet midden in een segment gekeken te worden of de variabele nog wel mag leven.

Direkt hiermee in verband staat de creatie van een leeg segment voor ieder block-label. Meer over lokale variabelen en het gebruik van de extra segment-grenzen in 5.6.

De nieuwe segment-indeling ziet er dus als volgt uit. Een segment-grens wordt gelegd:

a) voor een statement, dat gelabeld is.

b) na een CALL, RTN, EXIT en JUMP.

c) na een statement, dat een CALL, RTN, EXIT of JUMP naar een label buiten het statement bevat.

d) na het laatste statement van een <block>.

e) na het label van een <block>.

In de gevallen b) en c) wordt bovendien het betreffende statement in de OUT-knoop gezet.

5.2.2 Definitieketens en control-flow-kanten.

Na de creatie van een knoop wordt bekeken of deze een segmentgrens tot gevolg heeft. In dat geval wordt hij in de OUT-knoop gezet. Dan worden de definitieketens aangepast. Als de knoop in de OUT-knoop staat, dan wordt de OUT-knoop het beginpunt van alle definitie-ketens. Is dat niet het geval, dan wordt de nieuwe knoop het voorlopig begin van de definitieketens van de variabelen, die de nieuwe knoop definieert. (Merk op, dat er voor iedere variabele in ieder segment een definitieketen ligt).

Als een segment afgesloten is, worden de exits van het segment bepaald. Dan worden de successors van deze exits en de predecessors van de entry opgezocht. En tenslotte worden de control flow kanten tussen de al bestaande segmenten uit de body en het nieuwe segment aangebracht.

5.2.3 Data-afhankelijkheids-kanten

Om de data-afhankelijkheids kanten te leggen wordt voor elke variabele, die het statement definieert, de voorgaande definitie opgezocht. Tussen beide knopen wordt dan een kant gelegd.

Als het statement de variabele definieert, dan worden bovendien de knopen opgezocht, die afhankelijk zijn van die voorlaatste definitie. Vanaf elk van die knopen (dat zijn de gebruiken, die tussen de voorlaatste definitie en het huidige statement liggen) wordt een kant naar het statement gelegd.

5.2.4 Data-flow-paden

In 5.1 is de schijn gewekt, dat meteen alle data-flow-paden vanaf een gebruik naar de bijbehorende definities gelegd worden. Er is echter voor gekozen de aanleg van paden al tijdens de bouw van de graaf te beginnen (zie ook hoofdstuk 11). Daardoor is niet de hele knopenverzameling van de graaf voorhanden, op het moment, dat een begin gemaakt wordt met een pad.

De knopen van de graaf worden gecreëerd tijdens het ontleden van het programma. Op het moment, dat van een statement een knoop gemaakt wordt, zijn alleen de statements, die zich in de sourcetekst voor het huidige bevinden, beschikbaar. De andere, die verderop in het programma staan, zijn nog onbekend. Een knoop wordt na zijn body gemaakt.

Omdat in het algemeen de body van een subroutine niet bekend hoeft te zijn op het moment, dat een CALL gevonden wordt, kan niet bepaald worden voor welke variabelen het een definitie of tranparant is. Dus zullen in de buurt van CALL's geen paden gelegd worden.

Stel nu, dat een knoop behorende bij een gebruik net gecreëerd is. Dan wordt eerst bekeken of de knoop in de OUT-knoop hoort. Tenzij de knoop een CALL bevat of is, wordt daarna vanaf de (OUT-)knoop dat gedeelte van het pad aangelegd, dat tussen de IN-knoop van het omvattende segment en het gebruik ligt.

Hiertoe wordt eerst een kant van de voorgaande definitie naar de nieuwe (OUT-)knoop gelegd. Vanaf die definitie wordt volgens de methode, die in 5.1.1 besproken is naar de IN-knoop toegewerkt. Dat betekent, dat wel paden in de body van niet-primitieve definities voorafgaand aan het gebruik gelegd kunnen worden.

Als alle bijbehorende definities voor het gebruik in het segment staan, is het pad dus meteen klaar. Als dat niet zo is, dan eindigt het pad voorlopig bij de IN-knoop. Om te zien welke paden nog niet afgemaakt zijn, hoeft dus alleen de IN-knoop van een segment bekeken te worden.

Later, wanneer de bouw van een statement-body voltooid is en het statement bevat geen CALL's, worden deze onafgemaakte paden binnen de body verder voortgezet. Deze fase wordt de globale fase genoemd.

In deze fase worden de segmenten van de body één voor één bekeken. De onafgemaakte paden van die segmenten worden voortgezet vanaf alle predecessors van de entry van het segment. Dit voortzetten gebeurt zoals in 5.1.2 besproken is. Merk op, dat, omdat de body geen CALL's mag bevatten, alleen 'normale' predecessors voorkomen.

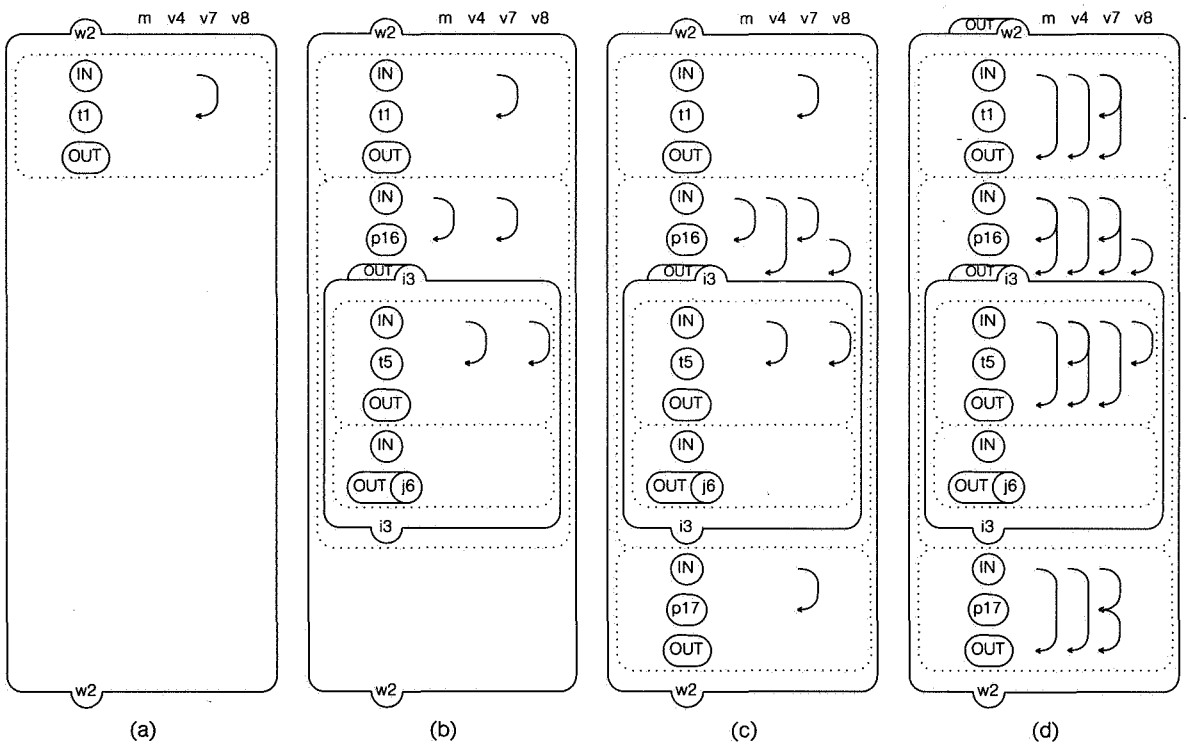
Analoog aan de onafgemaakte paden bij lokale aanleg, kunnen de onafgemaakte paden uit de globale fase gevonden worden door een entry te bekijken. In dit geval is dat de entry van de body, of met andere woorden de IN-knoop van het eerste segment. Zoals gezegd, geeft dat aan, dat het statement de variabele gebruikt. Het pad wordt dus op een hoger niveau weer voortgezet met eerst de lokale en dan de globale fase.

Tijdens de eerste fase worden paden dus aangelegd in afwisselend een lokale en een globale fase. Dit begint op het onderste nestingsniveau en gaat door tot op een niveau waarop een CALL staat. Vanaf dat niveau worden nog slechts lokale fasen uitgevoerd op alle knopen, behalve de OUT-knopen waarin zich een CALL bevindt.

5.2.5 Het voorbeeld

Om het bovenstaande te illustreren zal in figuur 4 de opbouw van de body van het while-statement uit het COMP_OTHERS segment uit het voorbeeld programma bekeken worden. Daarna in figuur 5 volgt een plaatje van de toestand waarin de graaf verkeert na de eerste fase. Het while-statement is hier natuurlijk weer in terug te vinden.

De variabelen, waarvoor geen kanten gelegd worden of zijn, zijn niet in figuur 4 weergegeven. Het grootste deel van de kanten uit de uiteindelijke graaf wordt dus later pas gelegd.



Figuur 4 De ontwikkeling van de w2-knoop van het voorbeeld-programma

a) Het while-statement is tot en met de test ontleed. Op de test is de lokale fase van het pad voor de lokale variabele v7 ('address') gemaakt. Het pad is nog niet klaar, omdat het eindigt bij de IN-knoop.

b) De ontleding is gevorderd tot en met het if-statement. In het if-statement is de globale fase al achter de rug. Het if-statement gebruikt dus de variabelen v4 en v8 ('guess' en de lokale variabele 'color'). Omdat de if-knoop een JUMP bevat wordt de knoop in de OUT-knoop gezet.

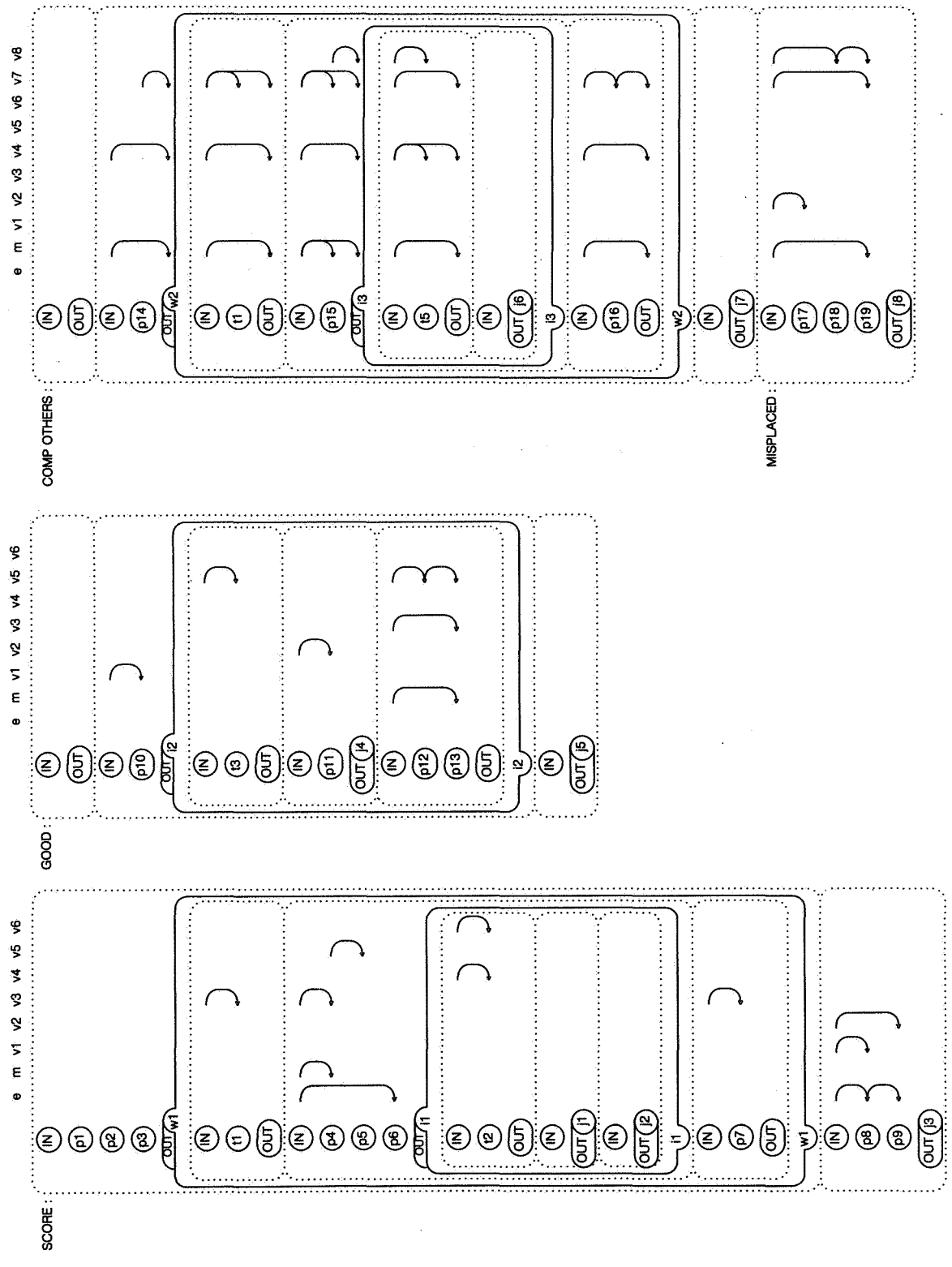
c) De situatie na de lokale fase voor de gebruiken van de if-knoop. Het derde segment is ook al gemaakt. De globale fase is nog niet ingegaan.

d) Tenslotte de situatie nadat de globale fase is afgesloten. De while knoop is klaar. Omdat de JUMP van het if-statement tot buiten het while-statement reikt wordt de while-knoop ook in de OUT-knoop gezet. Het while statement gebruikt de variabelen v7 ('address') en v4 ('guess').

In figuur 5 is het resultaat van de eerste fase te zien. Herkenbaar zijn de kanten in de w2 knoop, die net uitgebreid bekeken zijn. Vergelijking met figuur 1 doet vermoeden, dat er nog heel wat kanten gelegd moeten worden in de derde fase. Ook is de voorlopige segmentindeling goed te zien, hoewel één extra segment weggelaten is (leeg segment voor 'SCORE').

5.3 De tweede fase: Het verzamelen van informatie

In de tweede fase van het programma moeten de onbekende eigenschappen van de subroutines bepaald worden. In het bijzonder worden voor iedere subroutine een viertal gegevens vastgelegd:



Figuur 5. Graaf na de eerste fase.

a) welke segmenten van de programma-body tot de subroutine behoren.
b) welke CALLs de subroutine-body bevat.
c) welke primitieve definities de body bevat.
d) of de body EXIT's bevat.

Deze gegevens worden verzameld door vanaf de subroutine-entry de graaf in de richting van de control flow af te zoeken. De segmenten, die daarbij gepasseerd worden, worden gemerkt met de naam van de subroutine. Tegelijk wordt genoteerd welke variabelen in het betreffende segment gedefinieerd worden en welke CALLs de OUT-knoop bevat.

Ook hier moet opgepast worden voor lussen in het programma. Dit wordt gedaan door alleen successors te nemen van segmenten, die nog niet gemerkt zijn met de naam van de subroutine onder behandeling. (Segmenten kunnen deel uitmaken van meerdere subroutines. Er moet dus op de naam gelet worden.)

Op een gegeven moment komt dit speuren tot een einde, omdat geen exit van het segment een successor heeft. Alle exits bevatten dus een RTN, EXIT of impliciete EXIT (lege exit van laatste segment van het programma). Deze kunnen meteen genoteerd worden.

Met de gegevens b)-d) kan nu bepaald worden welke variabelen de CALLs definieren en of ze EXITs bevatten. Dit wordt gedaan door het bepalen van de transitieve afsluiting naar de onderlinge aanroepen.

5.3.1 Het voorbeeld

Als deze methode op het voorbeeld programma wordt toegepast, wordt het volgende gevonden:

	GOOD	COMP_OTHERS
segmenten	3de, 4de en 5de	6de, 7de, 8ste en 9de
CALLs	COMP_OTHERS	-
RTNs	j5	j7, j8
definities	v1 (good), v2 (misplaced), v3 (address), v5 (color), m, **v7, v8**	v7 (address lok.), v8 (color lok.), v2, m (memory)
bevat EXITs	neen	neen

Dikgedrukte items zijn toegevoegingen of veranderingen door het bepalen van de transitieve afsluiting.

Nu bekend is, welke subroutines EXIT's bevatten, kan bepaald worden, welke statements, die door de aanwezigheid van CALL's in een OUT-knoop geplaatst zijn, daar later weer uit verwijderd moeten worden. Ook kan vastgesteld worden welke variabelen deze statements definiëren. Deze gegevens worden hieronder gebruikt.

5.4 De derde fase: Paden afmaken

Na de tweede fase zijn alle gegevens beschikbaar, die nodig zijn om de overige kanten te leggen. Omdat nu de hele graaf al klaar is hoeft dat niet in fasen te gebeuren. Nu kunnen de paden aangelegd worden zoals in 5.1 besproken is.

Als het programma geen CALL's bevat, komt deze fase neer op een globale fase voor de programma-body.

Ieder segment uit de body wordt als volgt behandeld:

- Allereerst wordt bekeken of de body van het statement in de OUT-knoop CALL's bevat. Als dat het geval is, dan heeft geen globale fase in deze body plaatsgevonden. Ook is de lokale fase op het huidige niveau overgeslagen. Daar wordt eerst iets aan gedaan.
  - De body wordt behandeld, zoals hier beschreven is. (Er wordt dus recursief afgedaald, tot het laagste niveau waarop nog CALL's staan)
  - Daarna worden de gebruiken van de knoop in de OUT-knoop bepaald door de entry van het statement te bekijken. Voor die variabelen wordt vanaf die OUT-knoop een pad gelegd. Hierbij wordt natuurlijk de methode uit 5.1.1 gebruikt.
- Dan wordt bekeken welke onafgemaakte paden het segment bevat. (IN-knoop bekijken, dus)
- Dan worden deze paden vanaf de predecessors (en eventueel call's) van de IN-knoop voortgezet, zoals in 5.1.2 beschreven is.

5.4.1 Het voorbeeld

In figuur 6 is het resultaat getekend van deze derde fase. Dit behoeft verder geen commentaar.

5.5 De vierde fase: Verwijderen van segmentgrenzen

De segmentindeling, die in de eerste fase gehanteerd is, is voorlopig. Uiteindelijk moeten de segmenten natuurlijk ingedeeld zijn, zoals dat in 4.2 vastgelegd is. Merk op, dat er meer segmenten gemaakt zijn dan de bedoeling is. Er hoeven dus alleen segmenten aan elkaar geplakt te worden.

Het verwijderen van een segmentgrens bestaat uit twee stappen. Eerst wordt, als dat aanwezig, het statement uit de OUT-knoop van het eerste segment gehaald. Daarna worden de segmenten gecombineerd door de OUT-knoop van het eerste segment en de IN-knoop van het tweede segment weg te halen.

5.5.1 Uit de OUT-knoop

Eerst wordt het statement uit de OUT-knoop gehaald. Daarna worden de kanten van en naar het statement en de OUT-knoop veranderd. Daarbij kunnen drie gevallen onderscheiden worden. Deze kunnen weer in verschillende situaties verkeren. In figuur 7 is getekend hoe de i2-knoop van het voorbeeld uit de OUT-knoop verwijderd wordt. Merk op, dat 'tegelijk' binnen het if-statement de CALL uit de OUT-knoop gehaald is. Onze aandacht richt zich echter op de verwijdering van het if-statement.

1) Het statement definieert de variabele

   De volgende twee situaties kunnen zich dan onafhankelijk van elkaar voordoen:

   a) Er ligt een kant vanaf een eerdere definitie naar de OUT-knoop.

      Er wordt dan een kant van die definitie naar het statement gelegd. De kant naar de OUT-knoop wordt verwijderd. (m, v2)

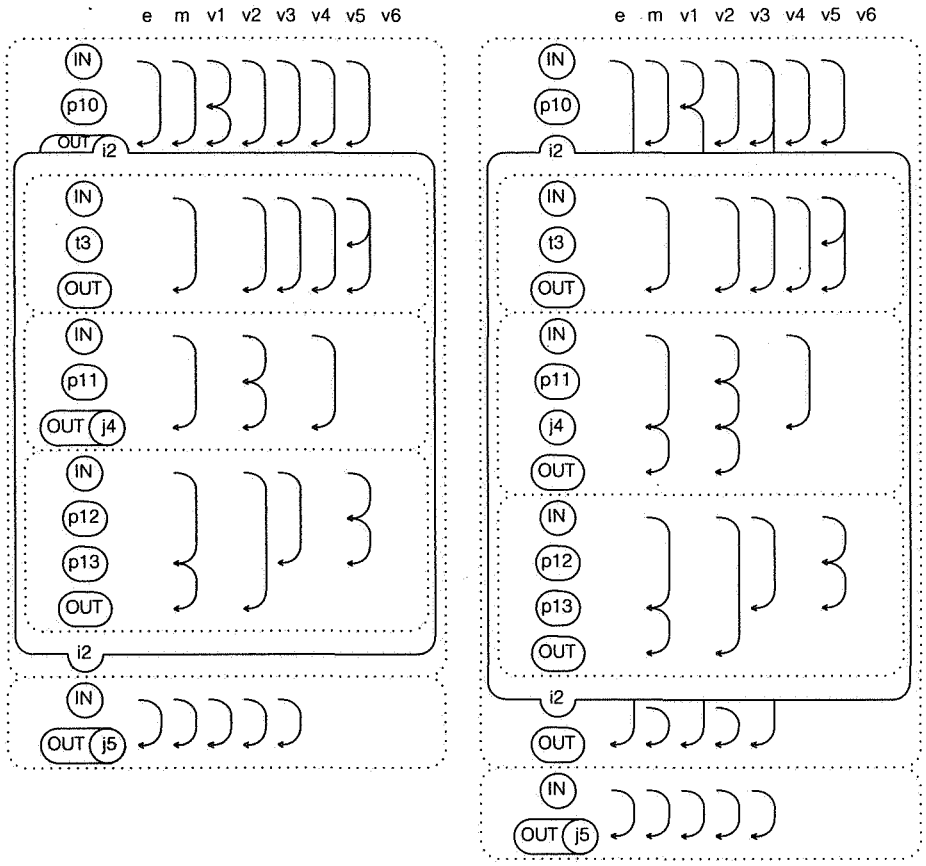
   b) Er ligt een pad vanaf de IN-knoop van het volgende segment.

      Er wordt een kant van het statement naar de OUT-knoop gelegd. (m, v2)

2) Het statement gebruikt de variabele, maar definieert hem niet.

   Er ligt dan in ieder geval een kant naar de OUT-knoop. De knoop waar deze vandaan komt, wordt hieronder de definitie genoemd. Er wordt een kant van deze definitie naar het statement gelegd. Als het pad zich in het volgende segment niet voortzet (vanaf de IN-knoop), dan wordt de kant van de definitie naar de OUT-knoop verwijderd. (v4, v5) Als dat niet zo is, dan blijft die kant gewoon liggen (v3)

Figuur 7. De graaf na de derde fase



Figuur 7 Een knoop wordt uit een OUT-knoop gehaald.

3) Het statement gebruikt noch definieert de variabele.

Er verandert dan niets aan de kanten. (e, v1)

Als dit alles achter de rug is worden ook de nodige d-a-kanten gelegd. Dit gebeurt analoog aan het d-a-kanten leggen in de eerste fase.

5.5.2 Segmenten aan elkaar plakken

Het eerste segment heeft nu of had al een lege OUT-knoop. Paden, die door die knoop lopen, worden allemaal vervolgd aan de IN-knoop van het tweede segment. Het enige, dat er te doen staat, is dus het kortsluiten van deze paden (kant leggen van de voorganger van de OUT-knoop naar de opvolger van IN-knoop) en daarna het verwijderen van de OUT-knoop en de IN-knoop.

Ook nu moeten weer enige d-a-kanten gelegd worden. Hetgeen weer gebeurt op de eerder beschreven manier.

5.6 Speciale gevallen

In 5.1 is een methode gegeven om d-f-paden te leggen. Daarbij zijn voor het gemak enkele gevallen, die een speciale behandeling vergen, buiten beschouwing gelaten. Nu een beeld is gegeven van hoe alles ongeveer in zijn werk gaat, lijkt het tijd om de puntjes op de i te zetten.

Er zijn vier van deze speciale gevallen. in de onderstaande secties zullen ze achtereenvolgens aan

bod komen. Eerst wordt getoond wat er mis gaat en dan wordt aangegeven hoe de methode uit 5.1 aangepast moet worden om problemen te voorkomen.

5.6.1 Gebruiken in subroutines

Als tijdens de aanleg van een pad een gelijk-gelabeld pad aangetroffen werd, mocht tot nog toe aangenomen worden, dat dat pad verder dezelfde kanten omvat, als het nieuw aan te leggen pad. Hetgeen een voldoende reden is om het leggen van paden in die richting verder te staken.

Nu zou dat stopcriterium bij gebruiken in subroutine-body's problemen kunnen veroorzaken. Dit wordt veroorzaakt, doordat er nu twee methoden zijn, volgens welke kanten gelegd worden. Ieder van die methoden heeft zo zijn eigen idee van wat predecessoren zijn (zie vorige sectie). In een subroutine-body worden, zoals al eerder opgemerkt is, beide methoden toegepast. Als een gebruik zichtbaar is vanaf een CALL, dan worden de kanten gelegd tijdens het doorlopen van de subroutine-body, en als het gebruik zich in de subroutine-body zelf bevindt, dan worden de kanten in de programma-body gelegd.

De predecessors, die geaccepteerd worden bij het doorlopen van een subroutine, vormen een deelverzameling van de predecessors, die tijdens het doorlopen van programma-body's gebruikt worden. Het probleem doet zich dus uitsluitend voor, wanneer de eerste methode verantwoordelijk is voor het al aanwezige pad.

In figuur 8 wordt een voorbeeld gegeven van wat er mis gaat, als er geen speciale maatregelen worden getroffen. Het voorbeeld is ontdaan van alle overbodige statements. Eerst wordt getoond hoe het goed gaat en daarna hoe het mis kan gaan. De volgende stadia zijn getekend:

a) De uitgangs situatie na de eerste fase van het programma De testen gebruiken x niet.
b) Na het afmaken van de paden vanaf het tweede gebruik.
c) Na het vervolgens afmaken van de paden vanaf het eerste gebruik. Dit is de situatie die bereikt moet worden.
d) Weer de uitgangssituatie
e) Maar nu worden eerst de paden vanaf het eerste gebruik gelegd.
f) Paden leggen vanaf het tweede gebruik stokt meteen doordat de kant in het test-segment aangetroffen wordt. Resultaat is, dat er een kant te weinig gelegd wordt.
g) Komt in de volgende sectie aan bod.

In dit voorbeeld kan door het kiezen van de juiste volgorde van handelen het probleem omzeild worden. Er is echter eenvoudig een ander voorbeeld te bedenken, waarin een dergelijke verwisseling geen soelaas biedt. De oplossing zal dus gezocht moeten worden in een wijziging van het stopcriterium, zonder dat de aangename eigenschap, dat lussen vermeden worden, aangetast wordt.

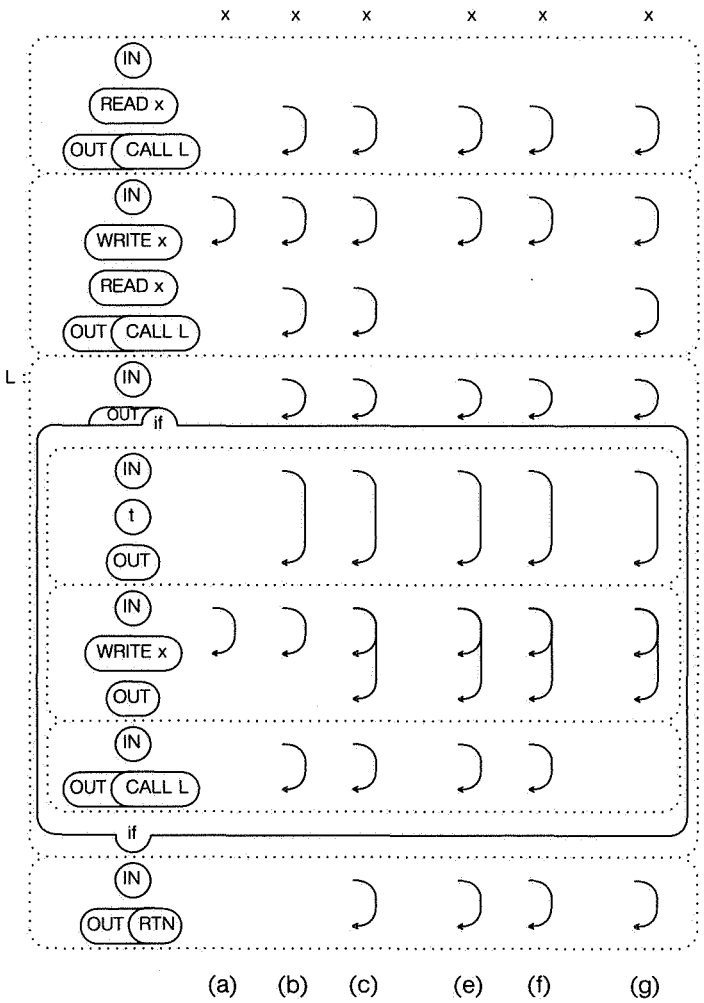
Een oplossing kan gevonden worden door het introduceren van een onderscheid tussen de kanten, die door beide methoden gelegd worden. De aanleg van kanten in de programma-body wordt dan alleen nog maar gestopt voor kanten van het goede type. Als er kanten van het verkeerde type liggen, dan worden deze vervangen door goede, maar wordt de aanleg niet gestaakt.

5.6.2 Transparant en recursief.

De vraag of een subroutine transparant is, wordt, net als bij statements, beantwoord door de body te doorlopen en te noteren of daarbij de entry bereikt wordt. Als tijdens dat doorlopen een recursieve aanroep aangetroffen wordt, wordt tijdelijk aangenomen, dat de subroutine niet transparant is, tenzij anders is aangetoond. Als achteraf blijkt, dat de subroutine wel tranparant is, dan is daar blijkbaar ten onrechte gestopt met het leggen van kanten.

In het voorbeeld uit de vorige sectie is voor het gemak buiten beschouwing gebleven, dat op een ander moment ook een keuze gemaakt wordt. Deze keuze moet worden gemaakt bij het overstappen van het laatste segment op het voorlaatste. Als daarbij eerst voor de else-tak gekozen wordt, wordt aangenomen, dat de subroutine niet transparant is, omdat dat nog niet aangetond is. Later blijkt echter dat deze aanname fout was, de subroutine is wel transparant. Er ontbreekt hierdoor een kant,

```
BEGIN
    READ   x
    CALL L
    WRITE   x
    READ   x
    CALL L
L:  IF   t
       WRITE   x
    ELSE
       CALL L
    ENDIF
    RTN
END
```



Figuur 8 Speciale gevallen

zoals in figuur 8 (g) te zien is.

Om deze fout te voorkomen moet, als blijkt dat een subroutine transparant is, de CALL's opgezocht worden, waar het pad ten onrechte beeindigd is. Deze CALL's bevinden zich in de subroutine, die deel uit maken van de recursieve component van de subroutine. Als een pad eindigt bij zo'n CALL, dan moet het pad vanaf die CALL binnen de subroutine-body voortgezet worden.

De recursieve component is de verzameling van subroutines, die elkaar onderling recursief aanroepen (de recursieve component van een subroutine is dus leeg, als de subroutine niet (indirect) recursief is). Het aanroepen van een van die subroutines, kan dus de aanroep van al die subroutines tot gevolg hebben. Recursieve componenten worden tegelijk met de transitieve afsluiting bepaald.

5.6.3 Tijdelijke segment-grenzen

Als een statement in een OUT-knoop is beland, maar daar volgens de segmentindeling uit 4.2. niet in thuis hoort, dan zal de segmentgrens aan het einde van de analyse opgeheven worden. Het statement komt dan midden in een segment te staan. In statement-body's, die niet in een OUT-knoop staan, liggen alleen kanten, als het statement die variabele definieert. Dus als het statement tussen een definitie en een gebruik van een variabele, die het zelf niet definieert, staat, dan mag er geen pad voor die variabele aan de exit beginnen.

Doordat de knoop tijdens de aanleg van de paden in een OUT-knoop staat, zouden dergelijke paden wel gelegd worden. In het voorbeeld in figuur 7 zou bijvoorbeeld de i2-knoop vol kanten gezet zijn voor de variabele 'e'. Met als gevolg, dat deze kanten bij het verwijderen van de knoop uit de OUT-knoop weer weggehaald zouden moeten worden.

Om dit te voorkomen moet eerst vastgesteld worden of een knoop in aanmerking komt om verwijderd te worden. Dit is het geval, als

1) het statement alleen exits heeft, die leeg zijn of een CALL bevatten.
2) geen van de CALL's in het statement een subroutine aanroept, die een EXIT bevat. Verder moet nagegaan worden, welke variabelen het statement definieert. Daarbij moet rekening gehouden worden met de variabelen, die de CALL's definieren. Al deze gegevens zijn beschikbaar na de tweede fase.

Stel nu, dat een knoop gevonden wordt, waarin geen kanten gelegd zouden moeten worden. Dan wordt de stap van het leggen van kanten in de knoop overgeslagen. In plaats daarvan wordt in alle subroutines, waarvan zich een CALL in het statement bevindt, een pad gelegd. Dit mag, omdat alle exits van het statement dezelfde successor hebben en dus iedere CALL bereikbaar is vanaf die exits.

De subroutines en het statement zelf zijn transparant, want ze definieren de variabele niet. Dus na aanleg van de paden in de subroutine-body's, kan het pad vanaf de OUT-knoop voortgezet worden.

5.6.4 Lokale variabelen

Lokale variabelen bestaan alleen binnen het blok waarin ze gedefineerd zijn. Als ze niet goed geinitialiseerd worden kunnen hun paden buiten het blok terecht komen. Om dit te voorkomen, moeten enige speciale maatregelen genomen worden.

Allereerst is er bij de voorlopige segmentindeling voor gezorgd, dat een blok in de graaf gerepresenteerd wordt door een integraal aantal segmenten. Ook is een eventueel label van een <block> in een leeg segment terecht gekomen. Nu kan voor ieder segment vastgelegd worden welke variabelen er bestaan. In het lege 'label' segment mogen de lokale variabelen niet bestaan.

Tijdens het leggen van paden voor lokale variabelen worden dezelfde procedures gehanteerd, als bij globale variabelen, alleen wordt het leggen gestaakt, als er een segment bereikt wordt, waarin de betreffende variabele niet bestaat. Net als bij het leggen van paden in subroutine-body's komt dit dus neer op een beperking van de acceptabele predecessoren van de bereikte entry's.

6 CONCLUSIES

In dit hoofdstuk zullen afbeelding en algoritme geëvalueerd worden. Hierbij zal gekeken worden naar de toepasbaarheid, andere bestaande methoden en eventuele verbeteringen en uitbreidingen. Omdat de graaf en het algoritme niet echt afhankelijk van elkaar zijn, zullen ze, respectievelijk in 6.1 en 6.2, apart besproken worden.

In sectie 6.1 wordt de graaf eerst aan de in de inleiding genoemde eisen getoetst. Dan wordt getoond, dat de afbeelding ook in andere codegeneratie-systemen en debuggers van nut kan zijn. Een andere, voor een soortgelijk project ontworpen, graaf komt in 6.1.2 aan de orde en tenslotte wordt in 6.2.3 bekeken hoe de graaf nog verbeterd kan worden.

Sectie 6.2 begint met een korte beschouwing van het algoritme. In 6.2.1 wordt nagegaan hoe het algoritme verder toegepast kan worden. Daarna wordt de snelheid van het algoritme geschat. Dan wordt de methode in 6.2.3 vergeleken met enkele andere data-flow-analyse methoden, zowel op toepasbaarheid voor de generatie van de graaf, als op snelheid. Als laatste komen eventuele verbeteringen van het algoritme aan de orde.

Tenslotte worden in 6.3 de resultaten op een rijtje gezet.

6.1 De afbeelding

De vraag is nu in hoeverre de graaf aan de gestelde eisen voldoet. Daartoe wordt bekeken hoe de gevraagde gegevens uit de graaf afgeleid kunnen worden:

- De oorspronkelijke struktuur van het programma

  Een eenvoudige afbeelding kan uit de graaf het source-programma reconstrureren (afgezien van de precieze volgorde).

- De mobiliteit van statements

  De mobiliteit van de statements is af te leiden uit de segmentindeling en de data afhankelijkheids kanten.

- De levensduur van variabelen

  De levensduur van variabelen is weergegeven met de data flow kanten.

De graaf bevat dus de gevraagde gegevens en deze zijn bovendien eenvoudig aan de graaf te onttrekken. (Dat laatste is in 4.6 in meer detail besproken.)

6.1.1 Andere toepassingen voor de graaf

De gegevens, die in de graaf aanwezig zijn, kunnen natuurlijk ook in andere codegeneratie-systemen gebruikt worden. Hoewel een compiler zich niet direct met parallellisme bezig zal houden, kan de informatie over de verschuifbaarheid van statements gebruikt worden om de levensduur van variabelen te verkorten, waardoor misschien minder registers tegelijk gebruikt hoeven te worden. Daarbij kan het handig zijn, dat, zoals in de inleiding al is gezegd, de geldigheid van de gegevens niet door het verschuiven van statements wordt aangetast. Hiermee onderscheidt de graaf zich van de gebruikelijke resultaten van een data-flow-analyse.

(Verschuivingen over de in de graaf aangegeven grenzen daarentegen maken de data-flow-kanten wel ongeldig. Een voorbeeld hiervan is het in de inleiding genoemde verwijderen van statements uit lussen. Bij de bespreking van het algoritme wordt ingegaan op de mogelijkheid de graaf aan dit soort veranderingen aan te passen. De codegenerator blijft echter verantwoordelijk voor de correctheid van de verschuiving)

De graaf kan ook in een debugger als programma-representatie gebruikt worden. Interpretatie van de graaf is eenvoudig realiseerbaar en de data flow gegevens die in de graaf aanwezig zijn, kunnen gebruikt worden voor het traceren van fouten. Bijvoorbeeld om te bepalen, welke variabelen door welk statement gedefinieerd zijn, of om het verloop van de waarden van variabelen na te gaan.

Ook kan de graaf beschikbaar gesteld worden aan de programmeur. Zoals uit de plaatjes blijkt wordt een overzichtelijk beeld van de gegevensstromen in het programma gegeven. Dit kan vergeleken worden met het idee, dat de programmeur zich hiervan vormt. (Dit voorstel tot het gebruik van de graaf vindt zijn oorsprong in persoonlijke ervaringen opgedaan tijdens het tekenen van de voorbeelden. Het vermeende gemak kan natuurlijk vertekend zijn door de aanwezige kennis van hetgeen in de graaf verwacht mag worden)

Behalve de in de inleiding genoemde voordelen is er nog een voordeel aan het gebruik van een geneste graaf verbonden. Doordat de struktuur van de graaf dicht bij die van het source-programma staat, kunnen meldingen, die tijdens de analyse gegenereerd worden, in termen van het source-programma gegeven worden. Dit is natuurlijk vooral handig in een debugger.

6.1.2 Een andere, soortgelijke graaf

Wood beschrijft in [Wood79] een graaf, die ook gegevens levert voor een microcodegenerator. Deze generator is echter aan een bepaalde machine gebonden. Van de programmeur wordt verwacht, dat hij/zij zelf de registerallocatie uitvoert. Hierdoor is er natuurlijk geen behoefte aan data flow gegevens, dus ontbreken de data-flow-kanten in die graaf. De creatie van de graaf wordt hierdoor veel eenvoudiger.

Ook worden labels en sprongopdrachten anders behandeld. Alle statements, die op een gegeven niveau op zo'n punt volgen worden een niveau omhoog geduwd. Als er meerdere labels staan worden er dus meerdere niveaus aangemaakt. Om te voorkomen, dat statements voorafgaand aan deze punten over dat niveau heen verplaatst worden, worden ook nog kanten gelegd vanaf die knopen naar het nieuw aangelegde niveau. Op niveaus hoger dan dat, waarop het label of de sprong staat, wordt het omvattende statement gefixeerd met alleen kanten. Minder fraai hiervan is, dat hetzelfde probleem in de graaf op twee verschillende manieren tot uitdrukking gebracht wordt.

6.1.3 Verbeteringen aan de afbeelding

Verbeteringen aan de afbeelding hebben tot doel de codegenerator meer vrijheid te geven in de volgorde waarin de statements behandeld worden. De verwachting is immers, dat een grotere bewegelijkheid van de statements, efficiëntere microprogramma's oplevert. De bewegelijkheid wordt beperkt door twee factoren: de segmentgrootte en de data afhankelijkheid. Vergroting van segmenten is mogelijk, hoewel de mogelijkheid daartoe ook kan duiden op een slechte programmeerstijl. Van het verminderen van de data afhankelijkheid, op het eerste gezicht misschien onmogelijk, wordt de grootste verbetering verwacht.

Segmentgrootte

Segmenten kunnen vergroot worden door bestaande segmenten waar mogelijk aan elkaar te plakken. Dit is toegestaan als alle exits van een segment één en dezelfde opvolger hebben en die opvolger alleen die exits als voorganger heeft. Dus, dat het tweede segment hoe dan ook volgt op het eerste en andersom. (Een EXIT of RTN heeft geen opvolger, dus mag niet in het eerste segment staan)

Een voorbeeld hiervan is het gebruik van een JUMP als 'break'-statement. Dat wil zeggen, dat een JUMP gebruikt wordt om de executie van een (aantal geneste) gestruktureerd(e) statement(s) te onderbreken. (Volgens sommigen het enige te rechtvaardigen gebruik van sprongopdrachten in een gestruktureerde programmeertaal) In figuur 9 wordt hiervan een eenvoudig voorbeeld gegeven.

Het WHILE-statement hoeft niet in de OUT-knoop te staan en als er niet van elders naar L gesprongen wordt kan de segmentgrens verwijderd worden. De JUMP moet dan in de graaf wel vervangen worden door een nieuw in te voeren BREAK-knoop (S_YALLL hoeft dus niet gewijzigd te worden). Het label wordt eventueel met de grens meeverwijderd. Binnen het WHILE-statement is de segmentgrens na de JUMP natuurlijk niet te vermijden. Dus alleen op het niveau van het label verandert er iets.

```
    WHILE true
         some_primitives
         IF test
              JUMP L
         ENDIF
         some_primitives
    ENDWHILE
L:  statements
```

Figuur 9 Een break-achtige JUMP

Er zijn andere gevallen te bedenken waarin segmenten samengevoegd kunnen worden. Meestal zijn dit soort gevallen echter te wijten aan een slechte programmeerstijl (onnodige jump's en/of label's).

Data afhankelijkheid

De afhankelijkheid van statements kan gereduceerd worden door het herbenoemen van variabelen. Daarmee wordt bedoeld, dat een variabele, die na gebruik opnieuw gedefinieerd wordt een andere naam krijgt. In figuur 10 wordt hiervan een voorbeeld gegeven.

voor:	na:
`ADD x, y, 1`	`ADD x, y, 1`
`MUL y, x, x`	`MUL y, x, x`
`MOV x, 2`	`MOV a, 2`
`ADD z, x, x`	`ADD z, a, a`

Figuur 10 Herbenoeming van een variabele.

Het resultaat van de herbenoeming in figuur 10 is, dat de afhankelijkheid tussen het tweede en het derde statement vervalt, waardoor er opeens zes toegestane volgordes zijn in plaats van één.

6.2 Het algoritme

In deze sectie komt niet alleen het algoritme aan de orde, maar ook de data-flow-analyse methode die er aan ten grondslag ligt.

Deze methode gebruikt de control flow in het programma om de volgorde te bepalen, waarin de knopen van de graaf in behandeling worden genomen. Het komt er op neer, dat vanaf een gegeven punt alle control flow paden afgelopen worden tot het doel bereikt is. Hierdoor is de methode onafhankelijk van de vorm van de control flow graaf. Dit kan een voordeel zijn bij een analyse op een programma's, waarin tamelijk willekeurige sprongen toegestaan zijn.

6.2.1 Andere toepassingen

Hoewel voor de productie van de graaf de control flow op een nogal speciale manier wordt doorlopen, is het natuurlijk mogelijk het onderliggende principe voor andere data-flow-analyses te gebruiken. Ook maakt het niet uit in welke richting de paden afgelopen worden. Hetgeen in hoofdstuk 5 getoond is, is dus een speciale toepassing van een algemene data-flow-analyse methode.

Het lijkt mogelijk dezelfde methode te gebruiken om een bestaande graaf aan te passen aan door de code-generator of de programmeur aangebrachte wijzigingen. Natuurlijk is iedere methode

hiertoe in staat (door de analyse helemaal over te doen), maar dat kan veel tijd kosten. De hoeveelheid werk, die met gebruik van de hier gepresenteerde methode verricht moet worden, hangt voornamelijk af van het karakter van de verandering. Een verandering, die slechts een kleine invloed heeft op de gegevensstromen in het programma, kan snel verrekend worden.

6.2.2 De snelheid van het algoritme

Een belangrijke eigenschap van data-flow-analyse methoden is de snelheid waarmee de analyse wordt uitgevoerd. Met snelheid wordt de orde van de tijd bedoeld, die nodig is om de analyse uit te voeren, als functie van de grootte van het source-programma (de asymptotische tijdscomplexiteit). In het algemeen wordt de snelheid van een data-flow analyse methode uitgedrukt in N, het aantal control flow kanten. Omdat een knoop meestal een begrensd (in dit geval maximaal twee) control flow opvolgers heeft, is N evenredig met het aantal knopen en de grootte van het source-programma.

In deze sectie wordt een begrenzing gegeven voor de snelheid waarmee de graaf geproduceerd wordt. De schatting betreft dus alleen deze toepassing, hoewel aan het einde aangegeven wordt hoe het algoritme zich gedraagt bij andere data-flow-analyses.

Het zal blijken, dat de snelheid in de orde van NLlog(L) is, waarbij L het aantal tegelijk levende variabelen is. Dit aantal L zal aan een nader onderzoek onderworpen worden, hetgeen geen harde resultaten oplevert, maar wel een idee geeft van de samenhang met de grootte van het programma. In de volgende sectie zal blijken, dat, hoewel sommige methoden lineair genoemd worden, deze ook afhankelijk zijn van deze parameter.

Een onder- en een bovengrens voor de snelheid

Onderdeel van het algoritme is ook het aanleggen van definitie-ketens, control-flow-kanten en data-afhankelijkheids-kanten. Zonder dit aan te tonen nemen we aan, dat dit niet ingewikkelder is, dan het aanleggen van de data-flow-kanten. Het bepalen van de transitieve afsluiting wordt eveneens buiten beschouwing gelaten, omdat dit hoort tot de zogenaamde interprocedurele analyse, die bij snelheids beschouwingen van andere methoden ook meestal buiten zicht blijft. Of dit terecht is, is maar zeer de vraag.

Om de snelheid van de methode te bepalen, wordt eerst de hoeveelheid werk per kant bepaald. Later zal geschat worden hoeveel kanten er gelegd moeten worden. Overigens is het aantal kanten bepaald door de specificatie van de afbeelding. Iedere methode om de graaf te maken zal die kanten op een gegeven moment moeten genereren. Deze complexiteit wordt bepaald door het probleem en is dus een ondergrens voor de snelheid waarmee de graaf gegenereerd kan worden.

Bij de beschouwingen worden twee moeilijk in te schatten aspecten van de paden aanleg buiten beschouwing gelaten. Ten eerste is dat het feit, dat de nesting alleen afgedaald wordt, als daar definities aangetroffen zullen worden. Ten tweede zijn dat de overlevingspaden door subroutines, die een variabele niet definieren (zie ook 4.5). Voor het gemak wordt maar aangenomen, dat de respectievelijke positieve en negatieve invloeden op de snelheid tegen elkaar wegvallen.

Als er een kant midden in een segment gelegd moet worden, wordt de definitie keten bekeken om de vorige definitie te vinden. Dan wordt de kant gelegd of, als er al een kant ligt, het kanten leggen gestopt. (In dat laatste geval wordt het werk aan de al aanwezige kant toegekend. Daar deze test slechts een beperkt aantal keren uitgevoerd wordt, betekent dit geen essentiële toename van het werk.) Om dit alles te doen moeten enkele tabellen worden doorzocht. De lengte van deze tabellen hangt af van de hoeveelheid in leven zijnde variabelen. Dit aantal wordt L genoemd. Later zal aannemelijk gemaakt worden, dat L, hoewel afhankelijk van de lengte van het programma, binnen een programma redelijk constant is.

Bij het aantreffen van een IN-knoop is een segment-overgang nodig. Hiervoor wordt niet in tabellen gezocht, omdat alle exits aan bod komen. Wel wordt de nesting afgedaald, maar het is redelijk aan te nemen, dat nestingsdiepte niet afhangt van de grootte van het programma. De hoeveelheid tijd, die in zo'n overgang gestoken wordt is dus niet afhankelijk van de grootte van het programma.

(Indien gewenst kan de sprong ook in één keer gemaakt worden, zodat de hoeveelheid werk zeker constant is)

Aangezien het doorzoeken van een tabel logaritmisch af kan hangen van de lengte van de tabel (binary search), wordt aangenomen, dat per kant de hoeveelheid werk O(log(L)) is. De totale hoeveelheid werk is dus O(Klog(L)), waarbij K het aantal te leggen kanten is. Dit aantal K is evenredig met de grootte van de graaf, N, en met het aantal variabelen, dat tegelijk in leven is, L. De snelheid van het algoritme is dus NLlog(L).

Nu is het de bedoeling de snelheid uit te drukken in N, dus moet L geschat worden. Allereerst kan opgemerkt worden, dat het totaal aantal variabelen een bovengrens voor L is. Al die variabelen moeten in instructies gebruikt en gedefinieerd worden, dus dat aantal zal maximaal O(N) zijn. Anderszijds lijkt het onzinnig aan te nemen, dat het aantal variabelen afneemt als het programma groter wordt, dus het is minimaal konstant. De snelheid van het algoritme ligt dus ergens tussen N en N^2log(N).

Het aantal tegelijk levende variabelen L

Bovenstaande levert een nogal grove begrenzing van de snelheid op. Daarom wordt nu een poging ondernomen iets meer te zeggen over L, het aantal tegelijk levende variabelen, of met andere woorden de breedte van de gegevensstromen in het programma.

Eerst wordt het begrip "lokaal gebruikte variabele" gedefinieerd.

Def. Een *lokaal gebruikte variabele* is een variabele, die slechts gedefinieerd en gebruikt wordt in een duidelijk begrensd gedeelte van een programma.

Hieronder vallen dus de lokale variabelen in procedures, parameter(-achtige) variabelen en variabelen, die gebruikt worden voor tussenresultaten. Het is onbelangrijk, hoe deze variabelen gedeclareerd worden (In S_YALLL zal door de on-orthodoxe behandeling van lokale variabelen een groot deel van de lokaal gebruikte variabelen globaal gedeclareerd worden). Andere variabelen zullen *globaal gebruikt* genoemd worden.

De nu volgende onbewezen stelling wordt ingegeven door de vergelijkbare stelling, dat de lengte van procedures bij grotere programma's redelijk constant is en de lengte van het programma terug te vinden is in een evenredige toename van het aantal procedures. Het lijkt logisch eenzelfde stabilisatie voor het gebruik van variabelen te veronderstellen.

Stelling van Reinier

Het aantal lokaal gebruikte variabelen, dat tegelijk leeft, is binnen een programma redelijk konstant en onafhankelijk van de lengte van het programma.

Het aantal lokaal gebruikte variabelen neemt dus lineair toe met de grootte van het programma. Omdat de levensduur van deze variabelen echter beperkt is, neemt het totaal aantal kanten voor deze variabelen ook lineair toe.

De hoeveelheid lokale variabelen in L wordt dus constant verondersteld. De toename van L wordt dus bepaald door de toename van het aantal globaal gebruikte variabelen. Hierover zijn moeilijk schattingen te maken, maar het ligt voor de hand, dat dit aantal ver achter blijft bij het totaal aantal lokaal gebruikte variabelen. Als het al O(N) is, dan zal de factor heel klein zijn.

Aanvullende opmerkingen

- De in de voorgaande secties afgeleide snelheid is die voor deze speciale toepassing. Het is niet mogelijk de snelheid van het methode onafhankelijk te zien van de toepassing, omdat die toepassing bepaalt hoe vaak en in hoeverre de graaf doorlopen wordt. Dit is specifiek voor deze methode.

(Als de methode gebruikt zou worden voor bijvoorbeeld een live/death analyse (d.w.z. per knoop wordt aangegeven of een variabele leeft) en er zou gebruik gemaakt worden van bit-arrays om aan te geven of een variabele leeft, dan zouden geen tabellen doorzocht hoeven worden. De snelheid zou dus NL zijn en daarmee misschien sneller dan andere bekende methoden (zie sectie 6.2.2).)

- De snelheid van de methode hangt af van de grootte van de graaf en de toepassing. Hiermee is de eerder genoemde onafhankelijkheid van de vorm van de control flow graaf aangetoond. De snelheid wordt dus niet beïnvloed, doordat de graaf irreducibel is, veel terugwaartse sprongen bevat of iets dergelijks.
- Tot nog toe is slechts gesproken over de orde van de snelheid. Dit hoeft echter niets te zeggen over de snelheid van de methode in de praktijk. Het kan zijn dat een lineare methode in de praktijk altijd langer bezig is, dan een exponentiële methode.

  Dit is vooral van belang in dit project, omdat een micro-geheugen over het algemeen klein is en dus slechts kleine programma's aankan. Het is moeilijk de snelheid in dit verband te schatten, dus wordt dat nagelaten.

6.2.3 Andere data-flow-analyse methoden

Een overzicht van andere data-flow methoden wordt gegeven in [Veen] en in [Kenn81]. In de laatste wordt ook uitgebreid op de werking ingegaan. Beweringen, die hier worden gedaan betreffen de methoden, die daar besproken worden. Er blijkt, dat andere methoden zich onderscheiden van de hier gepresenteerde, doordat de knopen van de graaf in een andere volgorde behandeld worden. Is hier de control flow bepalend, in de andere methoden is gekozen voor willekeur of de source-tekst volgorde.

De snelheid

De snelheid van data-flow methoden wordt in het algemeen gegeven als het aantal keren dat de graaf afgewerkt moet worden vermenigvuldigd met de lengte van de graaf. Er wordt verder niet gekeken naar de hoeveelheid werk, die per knoop verricht moet worden, hetgeen nodig is om tot een vergelijking te komen met de hier gepresenteerde methode. Deze hoeveelheid werk hangt af van het type analyse, dat verlangd wordt (misschien wordt het daarom meestal buiten beschouwing gelaten).

Per knoop is bij de meeste data-flow-analyses een of andere operatie op verzamelingen nodig. De snelheid van die operaties is evenredig met de grootte van de verzameling. Bij live/death-analyse is dit het aantal levende variabelen L. Een 'lineaire' methode heeft dan een snelheid NL. Bij gebruik van bit-arrays is de kardinaliteit van de verzameling gelijk aan het totaal aantal variabelen. Deze is O(N), dus de snelheid is $O(N^2)$, hetgeen trager zou zijn dan de hier gepresenteerde methode. De meeste methodes ondervinden overigens problemen met programma's met een of andere speciale control flow. Sommige kunnen dergelijke probleemgevallen niet aan, andere worden meestal een orde log(N) trager.

Andere methoden voor de constructie van de graaf.

Data flow analyse methoden laten zich volgens Veen en Kennedy onderverdelen in twee klassen, afhankelijk van de interne representatie van het source-programma. De eerste, meest onderzochte klasse, is die van de zogenaamde low level methoden. Het programma wordt bij deze methoden omgezet in een ongestruktureerde graaf, vergelijkbaar met de control flow graaf uit hoofdstuk 3. High level methoden gebruiken een soort parse tree als programma representatie. De hier gepresenteerde methode is dus een high level methode.

Low level methoden

Omdat een low-level-methode geen inzicht heeft in de nesting van de statements is hij niet in staat de graaf zonder meer op te leveren. Wel kan de graaf in twee fasen gebouwd worden. In de eerste fase zou dan een low-level-methode bijvoorbeeld een live/death-analyse uitvoeren, waarna in de tweede (high-level) fase met de resulterende gegevens de kanten gelegd worden.

In de eerste fase wordt een hoop overbodig werk gedaan, doordat er geen rekening gehouden kan worden met de nesting van statements. In de tweede fase is nog een analyse nodig van de gegevens uit de eerste fase, maar die kan wel vrij eenvoudig zijn.

Er zijn weinig lineaire low-level methoden. Een er van is de "graph grammar" methode van [FaKZ75]. Deze methode ontleedt de graaf aan de hand van een grammatica. Nadeel is, dat de methode alleen toepasbaar is op de door de grammatica bepaalde klasse van grafen. Hierdoor is de methode niet te gebruiken voor willekeurige S_YALLL-programma's.

Een andere 'meestal lineaire' methode is de "path compression" methode van [GrWe76]. Hierbij wordt een graaf door middel van transformaties gecomprimeerd tot één knoop. Alleen reducibele grafen kunnen behandeld worden, maar dat hoeft geen probleem te zijn, omdat een irreducibele graaf getransformeerd kan worden tot een reducibele. 'Meestal lineair' wil zeggen, dat de methode in het algemeen lineair is, maar voor sommige programma's N log(N) is.

High level methoden

High level methoden zijn (zoals blijkt) wel in staat de graaf direct te bouwen. Er zijn echter weinig methoden die bestand zijn tegen terugwaartse sprongen. De enige gevonden methode, die deze sprongen wel aan kan, is de 'method of attributes' beschreven in [BaJa78].

Dit is een iteratieve methode, waarin de graaf telkens in sourcetekst volgorde wordt doorlopen tot convergentie optreedt. De snelheid waarmee convergentie optreedt is afhankelijk van het aantal terugwaartse sprongen en while-statements. Aangetoond wordt dat als het programma alleen while-statements bevat twee passes van de graaf voldoende zijn. Bovendien zal wegens de beperkte nestingsdiepte van while-statements aannemelijk gemaakt kunnen worden, dat de methode 'lineair' is.

Deze methode mag dan ook in staat geacht worden de graaf zonder veel problemen te kunnen bouwen.

6.2.4 Verbeteringen in het algoritme

Er zijn geen voor de hand liggende methoden om de orde van de snelheid van het algoritme te verbeteren. Maar, zoals al eerder opgemerkt, de orde is niet zaligmakend. Dus worden hier enige suggesties gedaan, die intuïtief een verbetering lijken te geven, hoewel de orde soms verslechterd.

Een voor de hand liggende verbetering is te proberen het doorlopen van de graaf voor meerdere variabelen tegelijk te doen. Met een verzameling levende variabelen in de hand wordt dan de graaf afgezocht. Een voordeel zou zijn, dat het stappen in de graaf verminderd wordt. Helaas lijkt het niet mogelijk een strategie te bedenken, die kan garanderen, dat het (in pathologische gevallen) niet toch op de oude methode (het per variabele aflopen van de graaf) uitdraait. En dat terwijl het zoeken in tabellen gedeeltelijk vervangen wordt door het meer tijd kostende manipuleren van verzamelingen. Mischien kunnen tests uitwijzen of een dergelijke verandering zinvol is.

Een andere wijziging zou de indeling in passes kunnen betreffen. Nu worden in zowel de eerste als de derde pass kanten gelegd. Een beter overzicht wordt verkregen, door al het kanten leggen uit te stellen tot de derde fase. Dan kan ook de vierde fase naar voren geschoven worden tot voor de derde fase, waardoor de manipulaties van kanten in die vierde fase niet meer nodig zijn.

6.2.5 Mogelijke bijwerkingen

In hoofdstuk 5 zijn enige eisen aan het source-programma gesteld. Tegelijk is beloofd, dat tijdens de analyse afwijkingen hiervan gedetecteerd kunnen worden en eventueel gecorrigeerd. Daar wordt nu kort op in gegaan.

Syntactische correctheid

De kontrole op de syntactische correctheid vindt plaats tijdens het ontleden van het programma in de eerste fase. Als hierin fouten gevonden worden, dan zal de analyse moeten worden afgebroken. Dat wil zeggen, dat na het beëindigen van de eerste fase gestopt wordt. Andere fouten kunnen immers nog opgespoord worden.

Fouten in het gebruik van variabelen en constanten

Tijdens de eerste fase kan type-checking uitgevoerd worden. Ook hier leiden fouten tot het afbreken van de analyse.

Misbruik van labels

Tijdens de eerste fase wordt de control flow in de graaf weergegeven. Als blijkt, dat labels niet aanwezig zijn, illegale sprongen uitgevoerd worden of CALLs naar een label onder programma-niveau bestaan, dan blijkt dit tijdens de bouw van de graaf en kan de analyse afgebroken worden.

Dode code

Vergelijkbaar met de manier waarop de subroutine-body's bepaald worden, kunnen ook de programma- en statement-body's bepaald worden. Alles wat niet tot een body behoort is dode code. Deze gedeelten kunnen uit de graaf verwijderd worden.

Subroutines zonder terugweg

Als een subroutine een variabele niet definieert, dan moet hij transparant zijn voor die variabele. Stel nu, dat er voor een onbekende (dus zeker niet in de subroutine gedefinieerde) variabele een pad in de subroutine-body gelegd wordt. Dan moet de subroutine transparant zijn en moet het pad de subroutine-entry bereiken. Is dat niet het geval, dan is er geen RTN bereikbaar vanaf de entry en mogen alle CALL's van die subroutine zonder gewetensbezwaar vervangen worden door JUMP's naar de voormalige entry.

6.3 Samenvatting

In dit rapport zijn twee onderwerpen aan de orde gekomen. Ten eerste is dat een afbeelding van een programma op een graaf en ten tweede een nieuwe flow analyse methode.

De afbeelding voert een programma over in een graaf, waarin op een overzichtelijke en flexibele manier gegevens over dat programma worden weergegeven. Naast het oorspronkelijke programma bevat de graaf gegevens over de mobiliteit van de statements en de levensduur van de variabelen. De mobiliteit geeft een indruk van het parallellisme in het programma. De graaf is flexibel, omdat de gegevens bestand zijn tegen verschuivingen van statements binnen de aangegeven grenzen. De overzichtelijkheid vergemakkelijkt het gebruik van de graaf, waarbij ook gedacht mag worden aan het verstrekken van de graaf aan de programmeur.

De analyse-methode valt op door de benadering van het probleem. Doordat de control flow als leiddraad dient voor de analyse, is de snelheid onafhankelijk van die control flow. Dit in tegenstelling tot andere methoden. De snelheid kan daarentegen niet los gezien worden van de toepassing, hetgeen bij andere methoden wel kan en gebeurt. Om deze en andere methoden te vergelijken moet dus een toepassings afhankelijke analyse op de methoden verricht worden.

Voor de meeste toepassingen wordt de orde van de snelheid gegeven door: NLlog(L), waarbij L een maat is voor de hoeveelheid gegevens, waarmee gewerkt wordt. Bijvoorbeeld bij "live/death-

analysis" staat L voor het gemiddeld aantal levende variabelen, bij "available-expression-analysis" staat L voor het gemiddelde aantal beschikbare expressies. Andere methoden zijn ook afhankelijk van deze parameter. Voor 'lineaire' methoden lijkt een snelheid van O(NL) meestal haalbaar.

REFERENTIES

[BaJa78] W.H. Babich & M. Jazayeri, "Methods of Attributes for Data Flow Analysis", Acta Informatica **10**, pp 245-272 (1978)

[Born84] R. van den Born, "Implementatie voor een Struktuur-Behoudende Data Flow Analyse op S_YALLL-Programma's", supplement bij "Struktuur-Behoudende Data Flow Analyse op Programma's met GOTO-Statements", doktoraalskriptie, Universiteit van Amsterdam, (1984)

[FaKZ75] R.K. Farrow, K. Kennedy & L. Zucconi, "Graph Grammars and Global Program Flow Aanlysis", Proceedings of the 17th Annual IEEE Symposium on Foundations of Computer Science (nov 1975)

[FlRo81] G. Florijn & G. Rolf, "PGEN - a General Purpose Parsergenerator", IW 157/81, Mathematisch Centrum, Amsterdam (1981)

[GrWe76] S.L.Graham & M. Wegman, "A Fast and Usually Linear Algorithm for Global Flow Analysis", Journal of the ACM **23**(1), pp 172-202 (1976)

[Kenn81] K. Kennedy, "A Survey of Data Flow Analysis Techniques", pp 5-54 van "Program Flow Analysis - Theory and Applications", ed. S.S. Muchnick & N.D. Jones, Prentice Hall (1981)

[Klin82] P. Klint, "From SPRING to SUMMER", proefschrift, Amsterdam (1982)

[PaLT76] D. Patterson, K. Lew & R. Tuck, "Towards an Efficient, Machine-Independent Language for Microprogramming", Proceedings on the 12th Annual Workshop on Microprogramming, pp 22-35 (1979)

[Sint] M. Sint, "An overview of S_YALLL", niet gepubliceerd.

[Veen] A. Veen, "Flow Analysis", verschijnt als hoofdstuk 2 van "The application of Flow Analysis to the generation of Data Flow Code", proefschift, Amsterdam (publicatie vermoedelijk 1985)

[Wood79] G. Wood, "Global Optimization of Microprograms through Modular Control Constructs", Proceedings of the 12-th Annual Workshop on Microprogramming, pp 1-6 (1979)

APPENDIX DE SYNTAX VAN S_YALLL

De syntax van S_YALLL wordt gegeven in een uitgebreide Backus-Naur vorm. Hieronder volgt in het kort de betekenis van enkele symbolen. In [FlRo81] wordt een volledige definitie gegeven.

<naam>	een symbool
*	nul of meer herhalingen van het voorafgaande symbool
+	minstens één van het voorafgaande symbool
()	groepering van symbolen
{ s1 s2 }*	nul of meer herhalingen van s1. Als meer dan één s1, dan gescheiden door s2's. Kan ook met een '+' uitgevoerd worden.
[]	nul of één keer de inhoud
|	keuze

De LEXICAL's zijn symbolen, waarvan de syntax niet gegeven wordt.

```
LEXICAL identifier, integer, binary, mask, newl, eof .

<yalll_program>       ::=  <declarations>*
                           { <block> <newl> }* [ <eof> ] .

<declarations>        ::=    <stype> <identifier>
                             ( ',' [ <newl> ] <identifier> )*
                             <newl>
                           | <dtype> <identifier>
                             ( ',' [ <newl> ] <identifier> )*
                             <newl>
                           | 'CONST'  <identifier> '=' <constant>
                             ( ',' [ <newl> ] <identifier> '=' <constant> )*
                             <newl>    .

<stype>               ::=    'INTEGER'
                           | 'FLAG'    .

<dtype>               ::=    'BITS' <dimension>
                           | 'ONES' <dimension>
                           | 'TWOS' <dimension>
                           | 'SM'   <dimension> .

<dimension>           ::=  '[' <integer> ':' <integer> ']' .

<constant>            :=     [ '-' ]  <integer>
                           | <binary> .

<block>               ::=  [ <label> ] 'BEGIN' <newl>
                              <declarations>*
                              <labeled_statement>*
                           'END' .

<labeled_statement>   :=   [ <label> ] <statement> <newl> .

<label>               ::=  <identifier> ':' .

<statement>           ::=    <if_statement>
                           | <while_statement>
                           | <case_statement>
                           | <primitive> .
```

```
<if_statement>        ::=  'IF' <test> <newl>
                             <labeled_statement>*
                           ( 'ELIF' <test> <newl>
                             <labeled_statement>* )*
                           [ 'ELSE' <newl>
                             <labeled_statement>* ]
                           'ENDIF' .

<while_statement>     ::=  'WHILE' <test> <newl>
                             <labeled_statement>*
                           'ENDWHILE' .

<test>                ::=  <test_operand> <relop> <test_operand> .

<test_operand>        ::=  <identifier> [ '[' <integer> ']' ]
                           | <constant>
                           | <flag> .

<relop>               ::=  '=' | '~=' | '<' | '<=' | '>' | '>=' .

<case_statement>      ::=  'CASE' <selector> <newl>
                           ( 'OF' ( <key> ':' )+ <newl>
                             <labeled_statement>* )+
                           [ 'DEFAULT' ':' <newl>
                             <labeled_statement>* ]
                           'ENDCASE' .

<selector>            ::=    <identifier> [ <field> ]
                           | <flag> .

<field>               ::=  '[' <integer> [ ':' <integer> ] ']' .

<key>                 ::=  <identifier> | <constant> | <mask> .

<primitive>           ::=    <move>
                           | <integer_operation>
                           | <logical_operation>
                           | <arithmetic_shift>
                           | <external>
                           | <jump>    .

<move>                ::=  'MOVE' <two_addresses>  .

<integer_operation>   ::=  ( 'ADD' | 'ADD1' | 'SUB' | 'SUB1' )
                           <three_addresses>
                           [ <flags> ].

<logical_operation>   ::=    <monadic_logic_op>
                           | <dyadic_logic_op>
                           | <logic_shift>.

<monadic_logic_op>    ::=  'CMPL' <two_addresses> [ <flags> ] .

<dyadic_logic_op>     ::=  ( 'AND' | 'OR'   | 'XOR' )
                           <three_addresses>
                           [ <flags> ].
```

```
<logic_shift>              ::=( 'SRL' | 'SLL'  | 'SRC' | 'SLC' )
                           <two_addresses> ','
                           <count>
                           [ <flags> ].

<arithmetic_shift>    ::=  ( 'SRA' | 'SLA' )
                           <two_addresses> ','
                           <count>
                           [ <flags> ].

<two_addresses>       ::=  <destination> ',' <source>.

<three_addresses>     ::=  <destination> ',' <source> ',' <source>.

<flags>               ::=  '<' {<flag> ',' }+ '>' .

<flag>                ::=  'Z' | 'N' | 'C' | 'V' | 'P' .

<external>            ::=    'LOAD'  <destination> ',' <address>
                           |'STORE' <source> ',' <address>
                           |'READ'  <destination>
                           |'WRITE' <source>.

<jump>                ::=    'JUMP' <identifier>
                           |'CALL' <identifier>
                           |'RTN'
                           |'EXIT' .

<destination>         ::=  <identifier> .

<source>              ::=  <identifier> | <constant> | <flag> .

<count>               ::=  <identifier> | <constant> .

<address>             ::=  <identifier> [ ('+' | '-') <integer> ]
                           | <integer> .
```