21:35

1 (a)

De volgorde van de fasen is:

1) Lexical Analysis
2) Syntax Analysis
3) Semantic Analysis
4) Intermediate Code Generation
5) Machine Independent Code Optimization
6) Code Generation

21:39 7) Machine Dependent Code Optimization

(b) De output van de ene fase is de input van de volgende fase. Dan ziet de reeks van fasen, mét de bijbehorende invoer en uitvoer er als volgt uit:

Source program → [Lexical Analysis] → Token stream →
[Syntax Analysis] → Syntax tree → [Semantic Analysis] →
Syntax tree → [Intermediate Code Generation] → 3-adres code →
[Machine Independent Code Optimization] → 3-adres code →
[Code Generation] → assemby → [Machine Dependent Code Optimization]
21:46 → assembly.

14:47

2(a)

We kunnen dat concluderen, omdat de twee producties voor de variabele O allebei met de terminaal 'if' beginnen. Als we dan de variabele O moeten herschrijven, en we zien in de invoer de terminaal 'if' staan (we kijken één symbool vooruit), kunnen we niet beslissen welke van de twee producties voor O we moeten kiezen.

14:52

15:02

(b) Er komt in G geen links-recursie voor, want de rechterkanten van de producties voor M en O beginnen allemaal met een terminaal.

We moeten wel links-factorisatie toepassen, want beide producties voor O beginnen met dezelfde drie terminalen 'if expr then'

Daartoe introduceren we een nieuwe variabele O' voor de rest van beide rechterkanten.

We krijgen dan de grammatica G' met startvariabele S en de volgende producties:

$S \rightarrow M \mid O$

$M \rightarrow$ if expr then M else $M \mid$ id = num

$O \rightarrow$ if expr then O'

$O' \rightarrow S \mid M$ else O

15:10

(c) 15:13

	First	Follow
S	{if, id}	{$}
M	{if, id}	{$, else}
O	{if}	{$}
O'	{if, id}	{$}

15:18

(d) De top-down parsing table voor G' ziet er als volgt uit

	if	expr	then	else	id	=	num	$
S	S → M S → O	–	–	–	S → M	–	–	–
M	M → if expr then M else M	–	–	–	M → id = num	–	–	–
O	O → if expr then O'	–	–	–	–	–	–	–
O'	O' → S O' → M else O	–	–	–	O' → S O' → M else O	–	–	–

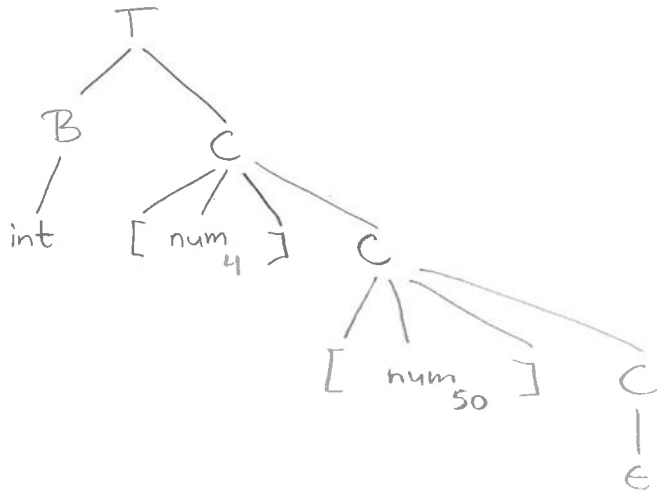
15:28

15:35

2(e) Nee, G' is geen LL(1) grammatica.
De top-down parsing table bevat namelijk diverse entries (combinaties van variabele en terminaal) met twee producties erin.

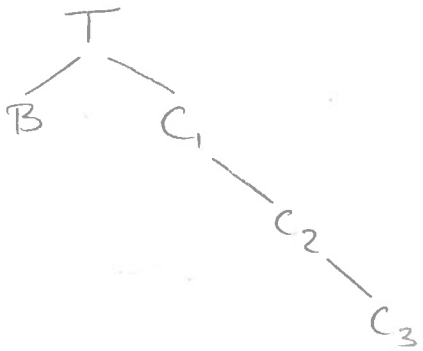
15:37
15:39 15:44

3 a)



15:47
15:56

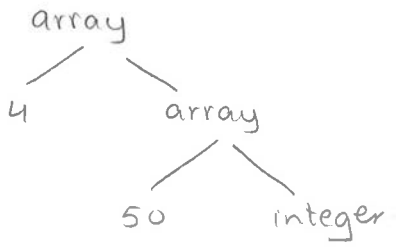
b) Ik nummer de variabelen C_i, zodat duidelijk is over welk voorkomen van C ik het heb:



B.type = integer
B.width = 4
t = B.type = integer
w = B.width = 4
C_3.type = t = integer
C_3.width = w = 4
C_2.type = array (50, integer)
C_2.width = 50 * 4 = 200
C_1.type = array (4, array (50, integer))
C_1.width = 4 * 200 = 800
T.type = array (4, array (50, integer))
T.width = 800

16:05

16:15
3(c)



16:16
16:~~16~~20

(d)

Nadat de variabele B in de productie $T \rightarrow BC$ is geparset, wordt de informatie over dit basistype opgeslagen in globale variabelen t en w, zodat we die informatie rechts onder in de afleidingsboom kunnen gebruiken.

De variabele C vertegenwoordigt één element van het basistype (helemaal rechts onder in de boom) of een array (van een array van, enz) zulke elementen.

De productie $C \rightarrow \epsilon$ wordt toegepast op de onderste C in de afleidingsboom. Die vertegenwoordigt dus één element van het basistype. Daarom worden de globale variabelen t en w (die over dit basistype gingen) hier in de attributen type en width van C gezet.

Bij de productie $C \rightarrow [\ num\ ]\ C_1$ wordt een array van num.value elementen van type C_1 gegenereerd. Daarom wordt

$C.type = array(num.value, C_1.type)$

en

$C.width = num.value \times C_1.width$.

Als we ten slotte ook de C in de productie $T \rightarrow BC$ geparset hebben, hebben we alle informatie over de type expressie (zowel het basistype als de dimensies van het array (als er inderdaad sprake is van een array) opgeslagen in de C (van onder naar boven doorgegeven). Het is derhalve voldoende om deze informatie naar T te kopieren (zowel type als width).

16:40
19:03

4(a)

De register descriptor voor een register bevat (tijdens het uitvoeren van een programma) de variabelen wier huidige waarde in dat register is opgeslagen.

De address descriptor voor een variabele bevat (tijdens het uitvoeren van een programma) de locaties (registers of geheugenlocaties) waar de huidige waarde van die variabele is opgeslagen.

19:08

(b)

In de address descriptor van y kunnen we zien of y al in een register zit. Dit is het geval, dan en slechts dan als er een register in deze address descriptor zit.

We kunnen de register descriptors gebruiken om te bepalen of er nog een register vrij is. Dit is het geval, dan en slechts dan als er een register is met een lege register descriptor.

19:13

(c) Wanneer y nog niet in een register zit en er ook geen register vrij is, zullen we een register 'vrij moeten maken!

Voor elk register bepalen we hoeveel 'stores' we nodig hebben om dat register vrij te maken. Daartoe lopen we alle variabelen v in de register descriptor van dat register af.

* als de huidige waarde van v ook nog op een andere locatie staat (volgens de address descriptor van v), dan is er voor v geen store nodig
* als de huidige waarde van v later in het programma niet meer nodig is (de variabele is op dat programma punt niet live), dan is er voor v geen store nodig.

19:21
18:55

* een speciaal geval hiervan is dat v gelijk is aan x, terwijl x niet ook gelijk is aan z, dat wil zeggen: terwijl x geen operand is van de + (het resultaat van de operatie)
* als de voorgaande situaties niet van toepassing zijn, zou v opgeslagen moeten worden, als we het register willen gebruiken voor y.

Voor elk register tellen we het benodigde aantal 'stores' (opsla-acties), en we kiezen een register met zo weinig mogelijk stores.

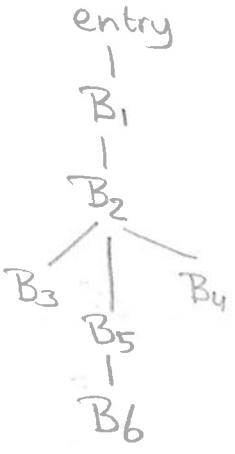
19:01

5 (a)

We noemen een knoop d in G een dominator van een knoop n in G, dan en slechts dan als ieder pad in G van de beginknoop entry naar n ook knoop d bevat.
In het bijzonder is iedere knoop n een dominator van zichzelf.

19:03

(b) De dominator tree voor G ziet er als volgt uit:

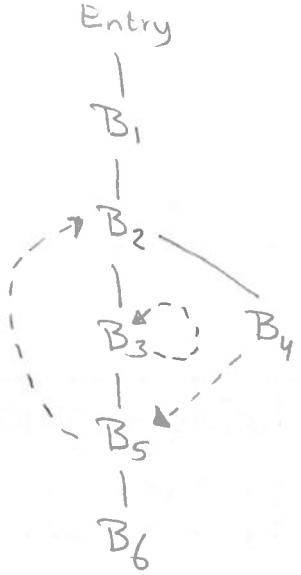


19:06

(c)

Een depth-first-spanning tree T voor G is

Als we de andere takken van G erbij tekenen:



19:10

(d)

Retreating edges: $B_5 \rightarrow B_2$, $B_3 \rightarrow B_3$ *

Cross edges: $B_4 \rightarrow B_5$

De back edges zijn in ieder geval retreating edges.
Omdat B_2 een dominator is van B_5, is $B_5 \rightarrow B_2$ inderdaad een back edge.
Omdat B_3 (per definitie) een dominator is van zichzelf, is ook $B_3 \rightarrow B_3$ inderdaad een back edge

* Retreating edges van G bij boom T zijn takken van een knoop naar een voorouder in de boom. Hierbij zien we elke knoop ook als voorouder van zichzelf. Zulke takken zijn er twee: $B_5 \rightarrow B_2$ en $B_3 \rightarrow B_3$
* Cross edges van G bij boom T zijn takken $n_1 \rightarrow n_2$ waarbij n_1 geen voorouder is van n_2, en n_2 ook geen voorouder is van n_1 in de boom T. Zo'n tak is er maar één: $B_4 \rightarrow B_5$

19:19

(e) Een depth-first ordering van de knopen in G, aansluitend bij boom T is een omgekeerde postorder ordening van de knopen.
Een postorder ordening van de knopen bij boom T is:

$B_6, B_5, B_3, B_4, B_2, B_1$, Entry

De bijbehorende depth-first ordering is (even omkeren:)

Entry, $B_1, B_2, B_4, B_3, B_5, B_6$

19:24

19:26

(f) een iteratief algoritme voor voorwaartse dataflow analyse, waarbij de knopen, iedere iteratie in de volgorde van een depth-first ordering bij T worden afgelopen, convergeren in hoogstens k+1 iteraties (of eigenlijk: in hoogstens k+2 iteraties, om de convergentie vast te stellen)

Laat k het maximale aantal retreating edges zijn op enig kring vrij pad in G. Dan zal (bij T)

19:34

De reden is dat de benodigde informatie voor de dataflow analyse zich in één iteratie (van het algoritme) over het pad kan voortbewegen tot aan de eerst volgende retreating edge. In de volgende iteratie kan die retreating edge gevolgd worden, alsmede het vervolg van het pad tot aan de volgende retreating edge. Enzovoort. In totaal heb je dus k+1 iteraties nodig om informatie van het begin tot het eind van het pad te laten stromen

19:40