TENTAMEN COMPILERCONSTRUCTIE
Donderdag 21 december 2017, 14:00 – 17:00 uur

Dit tentamen bestaat uit zes opgaven. waarbij steeds tussen [en] staat hoeveel punten er ongeveer mee te verdienen zijn. In totaal zijn er 100 punten te verdienen.

Als je het antwoord op een onderdeel niet weet, en je hebt dat antwoord nodig bij een later onderdeel, dan kun je het antwoord 'kopen' bij de docent.

Als er bij een opgave gevraagd wordt om uitleg of motivatie van je antwoord, is het belangrijk dat je die ook geeft.

---

1. [9 pt] Een compiler kan opgesplitst worden in een *front end* en een *back end*.

   (a) Welke fase(n) van de compiler vindt/vinden in ieder geval plaats in de front end? Welke fase(n) van de compiler vindt/vinden in ieder geval plaats in de back end?

   (b) Front end en back end communiceren met elkaar met behulp van een *intermediate representation*. Wat is het voordeel van zo'n intermediate representation bij het bouwen van compilers voor verschillende programmeertalen en verschillende *target machines*?

---

2. [10 pt] Beschrijf (duidelijk en volledig) een algoritme om bij een willekeurige grammatica G voor alle symbolen (terminalen en variabelen) X de FIRST-verzameling FIRST(X) te bepalen.

---

3. [23 pt] Beschouw de context-vrije grammatica G met startvariabele S en de volgende producties:

$$\begin{array}{lcl} S & \rightarrow & a\ A\ S \mid bb \\ A & \rightarrow & A\ d\ B \mid B \\ B & \rightarrow & b\ c\ b \mid b \end{array}$$

   (a) Leg uit waarom je zo al (zonder grondige analyse) kunt concluderen dat G geen LL(1) grammatica is.

   (b) Construeer vanuit G een context-vrije grammatica G' door

   - (indien van toepassing) links-recursie te elimineren
   - (indien van toepassing) links-factorisatie toe te passen.

   Leg uit hoe je G' uit G verkrijgt.

   (c) Bepaal voor elke variabele in de nieuwe grammatica G' zowel de FIRST- als de FOLLOW-verzameling.

   (d) Construeer de top-down *parsing table* bij de nieuwe grammatica G'. Wellicht ten overvloede: dit is dus iets anders dan de SLR parsing table.

   (e) Is de nieuwe grammatica G' een LL(1) grammatica? Motiveer je antwoord.

---

4. [22 pt] Tijdens het genereren van drie-adres code voor boolese expressies en *flow-of-control* instructies weten we bij goto-instructies vaak niet onmiddellijk waar we naartoe moeten springen. We kunnen *backpatching* gebruiken om dit achteraf op te lossen. Hierbij krijgt de variabele B in de grammatica (overeenkomend met een boolese expressie) attributen *truelist* en *falselist*. De variabele S (overeenkomend met een instructie) krijgt een attribuut *nextlist*.

   Naast deze variabelen gebruiken we hieronder hulpvariabelen M (met een attribuut *instr*) en N (met een attribuut *nextlist*).

   Beschouw het volgende *translation scheme* voor het genereren van de genoemde drie-adres code:

   $S \rightarrow \textbf{if}\ (B)\ M_1 S_1 N\ \textbf{else}\ M_2 S_2$
   $\{\ backpatch(B.truelist, M_1.instr);$
   $backpatch(B.falselist, M_2.instr);$
   $temp = merge(S_1.nextlist, N.nextlist);$
   $S.nextlist = merge(temp, S_2.nextlist);\ \}$

   $S \rightarrow \textbf{id}\ = \textbf{id}_1\ \textbf{binop}\ \textbf{id}_2;$ $\{\ S.nextlist = \textbf{null};$
   $gen(\textbf{id}.addr\ '='\ \textbf{id}_1.addr\ \textbf{binop}.op\ \textbf{id}_2.addr);\ \}$

   $B \rightarrow B_1\ \&\&\ M B_2$ $\{\ backpatch(B_1.truelist, M.instr);$
   $B.truelist = B_2.truelist;$
   $B.falselist = merge(B_1.falselist, B_2.falselist);\ \}$

   $B \rightarrow \textbf{id}_1\ \textbf{rel}\ \textbf{id}_2$ $\{\ B.truelist = makelist(nextinstr);$
   $B.falselist = makelist(nextinstr + 1);$
   $gen('\texttt{if}'\ \textbf{id}_1.addr\ \textbf{rel}.op\ \textbf{id}_2.addr\ '\texttt{goto}\ \_');$
   $gen('\texttt{goto}\ \_');\ \}$

   $M \rightarrow \epsilon$ $\{\ M.instr = nextinstr;\ \}$

   $N \rightarrow \epsilon$ $\{\ N.nextlist = makelist(nextinstr);$
   $gen('\texttt{goto}\ \_');\ \}$

   (a) Teken de afleidingsboom (*parse tree*) bij bovenstaande grammatica (met startvariabele S) voor het volgende 'programma':

   ```
   if (x>=a && x<=b)
     x=c-x;
   else
     x=x-c;
   ```

   (b) Pas bij de afleidingsboom van het vorige onderdeel de semantische acties toe zoals beschreven in het translation scheme. Geef bij elke variabele in de boom aan wat (de waardes van) de attributen (*truelist*, *falselist*, *nextlist* en/of *instr*) worden. Geef ook de resulterende drie-adres code. Ga ervanuit dat deze drie-adres code begint op instructienummer 100.

   (c) Leg uit wat er volgens het translation scheme gebeurt (de semantische actie) bij de productie

   $$S \rightarrow \textbf{if}\ (B)\ M_1 S_1 N\ \textbf{else}\ M_2 S_2$$

   Leg ook uit waarom dat gebeurt.

   N.B.: het gaat er hier om wat er in het algemeen gebeurt bij deze productie, en niet zozeer wat er gebeurt in het geval van ons voorbeeld'programma'.

5. [18 pt]

   (a) Wanneer noemen we een variabele x op een bepaald punt in een programma *live*?

   (b) Behalve naar liveness kunnen we ook specifieker kijken naar het *next use* van een live variabele x op een bepaald punt in een programma. Leg uit hoe we liveness- en next-use-informatie kunnen gebruiken voor *dead-code elimination* en voor registerallocatie.

   (c) a Binnen een *basic block* kunnen we systematisch next-use informatie bepalen door het block achterwaarts, instructie voor instructie door te lopen. Doe dit voor een basic block met de volgende drie-adres code:

```
1)  d=a+b
2)  e=d+e
3)  a=e
4)  c=a+b
5)  a=b-c
```

   Ga ervanuit dat alle vijf variabelen *live-on-exit* zijn. Geef de resulterende next-use informatie weer in een tabel van de volgende vorm:

	a	b	c	d	e
na regel 5 (on exit)	·	·	·	·	·
vóór regel 5					
vóór regel 4					
vóór regel 3					
vóór regel 2					
vóór regel 1 (on entry)					

   Gebruik hierbij de volgende notatie:

·	=	live, maar next use is niet bekend
–	=	niet live
i	=	next use in regel i

   Licht toe, waarom regel 1 uit de code de gevolgen voor de next-use informatie heeft die je in de tabel vermeldt.

Z.O.Z.

6. [18 pt] Om te controleren of een array `a` met integers op posities $0 \ldots n-1$ allemaal verschillende elementen bevat, kunnen we het volgende *brute force* algoritme gebruiken:

```
OK = true;
for (i=0;OK && i<=n-2;i++)
{ for (j=i+1;OK && j<=n-1;j++)
    if (a[i]==a[j])
      OK = false;
}
```

Als we aannemen dat een integer vier bytes in beslag neemt, kan een rechttoe-rechtaan vertaling van dit algoritme naar drie-adres code het volgende opleveren:

```
(1)  OK = true
(2)  i = 0
(3)  iffalse OK goto 20
(4)  t1 = n-2
(5)  iffalse i <= t1 goto 20
(6)  j = i+1
(7)  iffalse OK goto 18
(8)  t2 = n-1
(9)  iffalse j <= t2 goto 18
(10) t3 = 4*i
(11) t4 = a[t3]
(12) t5 = 4*j
(13) t6 = a[t5]
(14) iffalse t4 == t6 goto 16
(15) OK = false
(16) j = j+1
(17) goto 7
(18) i = i+1
(19) goto 3
(20) ...
```

(a) Bij het opsplitsen van drie-adres code in *basic blocks* maken we gebruik van *leaders*. Welke instructies in bovenstaande drie-adres code zijn de leaders?

(b) Teken de *flow graph* met de basic blocks bij bovenstaande drie-adres code. Nummer de basic blocks $B_1, B_2, \ldots$.

(c) Laat G een flow graph bij een programma zijn, met beginknoop *entry*. Wanneer noemen we een knoop d in G een dominator van een knoop n in G?

(d) Teken de *dominator tree* bij de flow graph uit onderdeel (b). Je hoeft niet de inhoud van elke knoop op te schrijven; werk gewoon met de namen $B_1, B_2, \ldots$.

(e) Wanneer noemen we een tak van een knoop a naar een knoop b in een flow graph G een *back edge*?

(f) Wat zijn de back edges in de flow graph uit onderdeel (b)?

(g) Geef bij elke back edge in de flow graph uit onderdeel (b) de bijbehorende *natural loop*.