HERTENTAMEN COMPILERCONSTRUCTIE
Donderdag 14 maart 2019, 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. [18 pt]

   (a) Geef een voorbeeld van een context-vrije grammatica met (precies) twee variabelen, waarin zowel directe als indirecte linksrecursie voorkomt. Wellicht ten overvloede: het is de bedoeling dat de taal van je grammatica niet leeg is.

   (b) Leg duidelijk uit hoe je in het algemeen de directe linksrecursie uit een context-vrije grammatica elimineert.

   In het boek wordt een algoritme beschreven om directe (*immediate*) en indirecte linksrecursie uit een context-vrije grammatica te elimineren, zonder de gegenereerde taal te veranderen. Dit algoritme begint ermee dat de variabelen in de grammatica geordend worden als $A_1, A_2, \ldots, A_n$ (als er n variabelen zijn).

   (c) Beschrijf (in woorden of pseudo-code, maar in ieder geval duidelijk en volledig) hoe dit algoritme verder gaat. Je mag ervan uitgaan dat de grammatica geen ϵ-producties en geen *cycles* bevat.

   (d) Pas je algoritme uit onderdeel (c) toe op je context-vrije grammatica uit onderdeel (a). Leg uit hoe je te werkt gaat en geef tussenresultaten.
   Mocht je grammatica uit onderdeel (a) toevallig wel ϵ-producties en/of cycles hebben, pas het algoritme dan gewoon toe, en zie wat er gebeurt.

2. [10 pt] Beschouw de context-vrije grammatica G met startvariabele S en de volgende producties:

$$\begin{aligned} S &\rightarrow AcB \mid bA \\ A &\rightarrow Bb \mid \epsilon \\ B &\rightarrow aS \end{aligned}$$

   De terminalen in G zijn dus a, b, c. Gegeven is dat de FIRST- en de FOLLOW-verzamelingen van de variabelen als volgt zijn:

X	FIRST(X)	FOLLOW(X)
S	$\{a, b, c\}$	$\{b, \$\}$
A	$\{a, \epsilon\}$	$\{b, c, \$\}$
B	$\{a\}$	$\{b, \$\}$

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

   (b) Parse de string *cab* met deze tabel. Laat bij iedere stap duidelijk zien wat je doet, bijvoorbeeld met behulp van een tabel van de volgende vorm:

Matched	Stack	Input	Action
	$S\$$	*cab*\$	...
...	...	...	...

3. [11 pt]

   (a) Wat verstaan we onder een *activation record*? Je hoeft hierbij niet alle soorten elementen van een activation record te benoemen.

   (b) Beschrijf vier soorten elementen van een activation record.

   (c) Beschouw de volgende C++-code voor sorteeralgoritme MergeSort:

```
void MergeSort (int links, int rechts)
{
  if (links!=rechts)
  { int mid = (links+rechts)/2;
    MergeSort (links, mid);
    MergeSort (mid+1, rechts);
    Merge (links, mid, rechts);
  }
}
```

   Hierbij

   - zitten de te sorteren getallen in een globaal array A,
   - geven de parameters `links` en `rechts` aan van welke index tot en met welke index in het array A door deze aanroep moeten worden gesorteerd,
   - schuift de hulpfunctie `Merge` de twee subarrays tussen indices `links` en `mid` en tussen indices `mid+1` en `rechts` in elkaar.

   Geef de *activation tree* met functie-aanroepen, behorend bij de aanroep `MergeSort (1,5)`.

---

4. [22 pt]

   (a) Om op lokaal niveau efficiënt registers te alloceren, kunnen we gebruik maken van *register descriptors* en *address descriptors*. Wat verstaan we onder deze twee 'descriptors'?

   (b) We willen nu de volgende drie-adres code omzetten in assembly:

```
a=b-d
d=b+c
b=e+f
e=a
c=d-c
```

   Gegeven is dat alle variabelen live-on-exit zijn.

   Zet de gegeven drie-adres code instructie-voor-instructie om in assembly, waarbij je **efficiënt** met registers omgaat. Je mag hierbij gebruik maken van

   - drie registers R1, R2 en R3, die aanvankelijk leeg zijn
   - assembly instructies van de volgende vorm:

```
ADD Ri, Rj, Rk
SUB Ri, Rj, Rk
LD Ri, x
ST x, Ri
```

     Hierbij zijn Ri, Rj en Rk registers (eventueel dezelfde), en is x een variabele. Bij ADD en SUB komt het resultaat in Ri te staan.

   Aan het eind hoef je de inhoud van de registers niet met allemaal store instructies veilig op te slaan.

   Geef, behalve de resulterende assembly, in een overzichtelijke tabel de inhoud van de register descriptors en de address descriptors

   - aan het begin
   - en na ieder stukje assembly dat met een drie-adres instructie overeenkomt.

   (dus in totaal zes keer).

   Geef bij iedere drie-adres instructie een korte motivatie van de keuze voor de gebruikte registers.

---

5. [9 pt]

   (a) Wat zijn *reaching definitions* op een bepaald punt in een programma?

   (b) Leg uit hoe we reaching definitions kunnen gebruiken

      i. om te controleren of een variabele die op een bepaald punt in het programma gebruikt wordt, mogelijk *undefined* is;

      ii. om vast te stellen dat een variabele op een bepaald punt in het programma een bepaalde, constante waarde heeft (handig bij constant folding).

---

6. [30 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 variabelen S (overeenkomend met een instructie) en L (overeenkomend met een lijst van instructies) krijgen een attribuut *nextlist*.

   (a) Leg voor elk van deze drie lijsten uit wat de lijst bevat.

   Naast deze variabelen gebruiken we hieronder een hulpvariabele M (met een attribuut *instr*). Beschouw nu het volgende *translation scheme* voor het genereren van de genoemde drie-adres code:

$S \to \{L\}$	$\{\ S.nextlist = L.nextlist;\}$
$L \to L_1 M S$	$\{\ backpatch(L_1.nextlist, M.instr);$ $L.nextlist = S.nextlist;\}$
$L \to S$	$\{\ L.nextlist = S.nextlist;\}$
$S \to \textbf{while}\ M_1\ (B)\ M_2\ S_1$	$\{\ backpatch(S_1.nextlist, M_1.instr);$ $backpatch(B.truelist, M_2.instr);$ $S.nextlist = B.falselist;$ $gen('\texttt{goto}\ M_1.instr');\}$
$S \to \textbf{id}_1 = \textbf{id}_2\ \textbf{binop}\ \textbf{num};$	$\{\ S.nextlist = \textbf{null};$ $gen(\textbf{id}_1.addr\ '='\ \textbf{id}_2.addr\ \textbf{binop}.op\ \textbf{num}.val);\}$
$B \to \textbf{id}_1\ \textbf{rel}\ \textbf{id}_2$	$\{\ B.truelist = makelist(nextinstr);$ $B.falselist = makelist(nextinstr + 1);$ $gen('\texttt{if}'\ \textbf{id}_1.addr\ \textbf{rel}.op\ \textbf{id}_2.addr\ '\texttt{goto}\ \_');$ $gen('\texttt{goto}\ \_');\}$
$M \to \epsilon$	$\{\ M.instr = nextinstr;\}$

   Hierin stelt het token **binop** een binaire operator voor, het token **num** een numerieke constante, en het token **rel** een relationele operator.

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

```
while (x<=y)
  x=x*2;
x=x/2;
```

   (c) 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. Vermeld de attributen in de volgorde waarin ze hun waarde krijgen. Geef ook de resulterende drie-adres code. Ga ervanuit dat deze drie-adres code begint op instructienummer 100.

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

$$S \to \textbf{while}\ M_1\ (B)\ M_2\ S_1$$

   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'.