HERTENTAMEN COMPILERCONSTRUCTIE
Dinsdag 7 maart 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. [6 pt] Noem drie redenen waarom het nuttig is om de lexicale analyse en de syntax analyse door *afzonderlijke* componenten van de compiler uit te laten voeren.

---

2. [7 pt] Beschouw de context-vrije grammatica G met startvariabele S en de volgende producties:

$$\begin{aligned} S &\rightarrow BA \\ A &\rightarrow \textbf{boolop}\ BA \mid \epsilon \\ B &\rightarrow \textbf{id}\ C \\ C &\rightarrow \textbf{relop id} \mid \epsilon \end{aligned}$$

Gegeven is dat de top-down parsing table bij grammatica G er als volgt uit ziet:

Non-terminal	Input Symbol			
	id	**boolop**	**relop**	$
S	$S \rightarrow BA$			
A		$A \rightarrow \textbf{boolop}\ BA$		$A \rightarrow \epsilon$
B	$B \rightarrow \textbf{id}\ C$			
C		$C \rightarrow \epsilon$	$C \rightarrow \textbf{relop id}$	$C \rightarrow \epsilon$

Parse de string **id relop id boolop id** 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\$	**id relop id boolop id**\$	output $S \rightarrow BA$
...	...	...	...

---

3. [26 pt] Beschouw de context-vrije grammatica G met startvariabele S en de volgende producties:

$$\begin{aligned} S &\rightarrow TB \\ T &\rightarrow aTb \mid ab \\ B &\rightarrow bB \mid b \end{aligned}$$

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

   (b) Bij het SLR parsen maken we gebruik van een LR(0)-automaat. De toestanden in deze automaat bevatten items van de (algemene) vorm $A \rightarrow \beta_1 \cdot \beta_2$.
   Leg uit wat het betekent wanneer we tijdens het SLR parsen in een toestand met daarin het item $A \rightarrow \beta_1 \cdot \beta_2$ zijn aangekomen.

   (c) Construeer de LR(0)-automaat bij grammatica G.

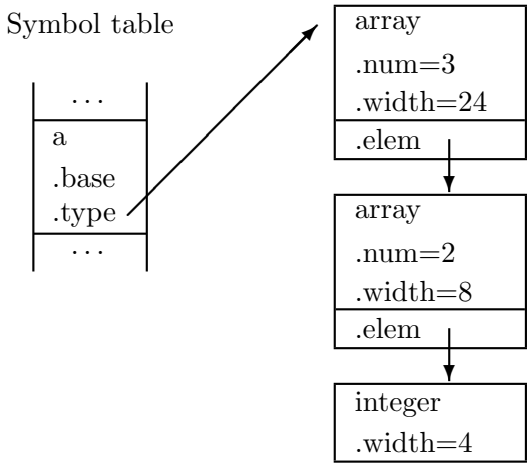
   (d) Construeer de SLR *parsing table* bij grammatica G.

---

4. [20 pt] Bij het parsen van de declaratie

```
int[3][2] a;
```

(`a` is een array van 3 bij 2 integers), kan een symbol table entry voor a ontstaan, die er schematisch als volgt uitziet:



Rechts in het plaatje staat een weergave van de syntax tree bij de type expressie $array(3, array(2, integer))$ van `a`. Beschouw nu de context-vrije grammatica G met startvariabele S en de volgende producties, om toekenningen van array-referenties te genereren:

$$\begin{array}{lcl} S & \rightarrow & \mathbf{id} = E \\ E & \rightarrow & \mathbf{id} \mid L \\ L & \rightarrow & \mathbf{id}[E] \mid L[E] \end{array}$$

(a) Geef (ad hoc) een afleidingsboom (*parse tree*) in G voor de instructie

```
c = a[i][j]
```

(b) Om toekenningen van array-referenties te vertalen naar drie-adres-code, geven we de variabele E een attribuut *addr* en geven we de variabele L drie attributen: *addr*, *array* en *type*.
Beschouw nu het volgende *translation scheme* voor het genereren van de genoemde drie-adres code:

$$\begin{array}{lcll} S & \rightarrow & \mathbf{id} = E; & \{\ gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr);\ \} \\ E & \rightarrow & \mathbf{id} & \{\ E.addr = top.get(\mathbf{id}.lexeme);\ \} \\ E & \rightarrow & L & \{\ E.addr = \mathbf{new}\ Temp(); \\ & & & \quad gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');\ \} \\ L & \rightarrow & \mathbf{id}\ [E] & \{\ L.array\ = top.get(\mathbf{id}.lexeme); \\ & & & \quad L.type\ = L.array.type.elem; \\ & & & \quad L.addr\ = \mathbf{new}\ Temp(); \\ & & & \quad gen(L.addr\ '='\ E.addr\ '*'\ L.type.width);\ \} \\ L & \rightarrow & L_1[E] & \{\ L.array\ = L_1.array; \\ & & & \quad L.type\ = L_1.type.elem; \\ & & & \quad t = \mathbf{new}\ Temp(); \\ & & & \quad L.addr\ = \mathbf{new}\ Temp(); \\ & & & \quad gen(t\ '='\ E.addr\ '*'\ L.type.width); \\ & & & \quad gen(L.addr\ '='\ L_1.addr\ '+'\ t);\ \} \end{array}$$

Pas bij de afleidingsboom uit onderdeel (a) de semantische acties toe zoals beschreven in het translation scheme. Geef bij elke variabele in de boom aan wat de attributen (*addr*, *array*, *type*, voor zover van toepassing) worden. Geef ook de resulterende drie-adres code.

Ga ervanuit dat benodigde nieuwe tijdelijke variabelen achtereenvolgens t1, t2, ... heten.

(c) Leg uit wat de betekenis is van het attribuut *type* van de variabele L. Leg ook uit wat zijn functie is bij de vertaling naar drie-adres code volgens bovenstaand translation scheme.

(d) Is het attribuut *type* van de variabele L een *synthesized* attribuut of een *inherited* attribuut? Motiveer je antwoord.

---

5. [20 pt] Stel dat we assembly code voor de drie-adres instructie $x = y + z$ willen genereren, waarbij x, y en z drie (niet per se verschillende) integer variabelen zijn. Dan moeten de argumenten (*operands*) van de operator + in een register geladen worden, voordat de operator kan worden toegepast.

   (a) Voor de selectie van een register voor een variabele kunnen we gebruik maken van *register descriptors* en *address descriptors*. Wat verstaan we onder deze twee 'descriptors'?

   (b) Leg uit hoe we de register descriptors en address descriptors kunnen gebruiken om te bepalen of y al in een register zit, en of er nog een register vrij is.

   (c) Beschrijf een algoritme om een register R_y voor y te bepalen wanneer y nog niet in een register zit en er ook geen register vrij is. Dit algoritme moet (minstens) dezelfde onderdelen bevatten als het algoritme in het boek, maar deze onderdelen hoeven niet per se in dezelfde volgorde te staan.

---

6. [21 pt] Om een array A met integers op posities $0 \ldots n-1$ te sorteren, kunnen we gebruik maken van InsertionSort. Een mogelijke implementatie van InsertionSort ziet er als volgt uit:

```
for (i=1;i<=n-1;i++)
{ tmp = a[i];
  j = i-1;
  while (j>=0 && a[j]>tmp)
  { a[j+1] = a[j];
    j--;
  }
  a[j+1] = tmp;
}
```

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

```
(1)  t1 = 4*i
(2)  tmp = a[t1]
(3)  j = i-1
(4)  iffalse j >= 0 goto 15
(5)  t2 = 4*j
(6)  t3 = a[t2]
(7)  iffalse t3 > tmp goto 15
(8)  t4 = j+1
(9)  t5 = 4*t4
(10) t6 = 4*j
(11) t7 = a[t6]
(12) a[t5] = t7
(13) j = j-1
(14) goto 4
(15) t8 = j+1
(16) t9 = 4*t8
(17) a[t9] = tmp
```

(a) Bij het opsplitsen van drie-adres code in *basic blocks* maken we gebruik van *leaders*. In welke gevallen noemen we (in het algemeen) een instructie in de drie-adres code een leader?

(b) Welke instructies in bovenstaande drie-adres code zijn de *leaders*?

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

(d) In het algemeen kunnen we drie-adres code met verschillende soorten transformaties optimaliseren. Leg voor elk van de volgende vier soorten transformaties uit wat ze inhoudt en wanneer ze kan worden toegepast. Illustreer de uitleg met behulp van stukjes voorbeeld drie-adres code. Deze voorbeeld drie-adres code hoeft niet per se afkomstig te zijn uit bovenstaande code.

- *local common-subexpression elimination*
- *global common-subexpression eliminiation*
- *copy propagation*
- *code motion*
