TENTAMEN COMPILERCONSTRUCTIE
Donderdag 20 december 2018, 14.00 – 17.00 uur

Dit tentamen bestaat uit vijf 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. [13 pt]

   (a) In een programmeertaal als C kunnen scopes genest zijn, en kunnen er per scope variabelen gedeclareerd zijn. Geef twee redenen waarom we dan een symbol table *per scope* zouden willen hebben.

   (b) Een mogelijke Java implementatie van de symbol table maakt gebruik van een klasse `Env`, die er als volgt uitziet:

```
public class Env
{ private Hashtable table;
  protected Env prev;

  public Env (Env p);
  public void put (String s, Symbol sym);
  public Symbol get (String s);
}
```

   Beschouw nu een programmeertaal, waarin (a) een programma een block is, (b) een block is opgebouwd uit (eerst) declaraties en (daarna) statements, en (c) een statement zelf ook weer een block kan zijn. Dan kunnen we het werken met een symbol table per scope beschrijven met het volgende stukje translation scheme:

$$
\begin{array}{lll}
\textit{program} \rightarrow & & \{\ \textit{top} = \textbf{null};\ \} \\
& \textit{block} & \\
\textit{block} \rightarrow & '\{' & \{\ \textit{saved} = \textit{top}; \\
& & \quad \textit{top} = \textbf{new}\ \textit{Env}(\textit{top}); \\
& & \} \\
& \textit{decls stmts } '\}' & \{\ \textit{top} = \textit{saved}; \\
& & \}
\end{array}
$$

   Waar staat de variabele *top* voor? Leg vervolgens uit wat er bij deze twee producties in de semantische acties gebeurt en waarom.

   (c) Leg in woorden uit hoe de methode `get` van de klasse `Env` werkt.

---

2. [32 pt] Beschouw de context-vrije grammatica G met startvariabele S en de volgende producties:

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

   De terminalen in G zijn dus a, b, c.

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

(b) Bepaal FIRST(AcB) en FIRST(Bb)

(c) Is G een LL(1) grammatica? Motiveer je antwoord. Maak daarbij ook gebruik van je antwoord op onderdeel (b).

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

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

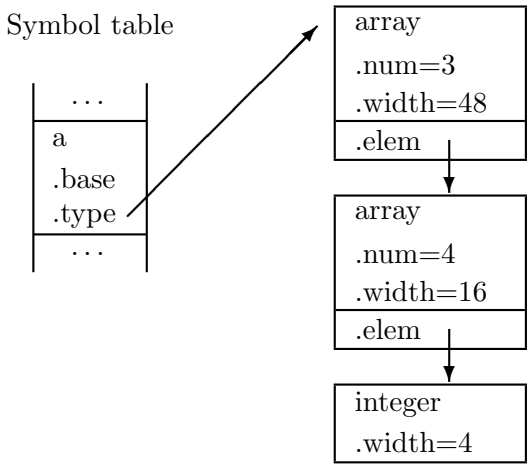
(f) Is G een SLR grammatica? Motiveer je antwoord.

---

3. [20 pt] Bij het parsen van de declaratie

```
int[3][4] a;
```

(`a` is een array van 3 bij 4 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 $array(3, array(4, integer))$ van `a`. Beschouw nu de context-vrije grammatica G met startvariabele S en de volgende producties, om toekenningen van array-referenties te genereren:

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

(a) Geef (ad hoc) een *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:

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

Pas bij de parse tree uit onderdeel (a) de semantische acties toe zoals beschreven in het translation scheme. Geef bij elke variabele in de boom aan wat de waarde van zijn attributen (*addr*, *array*, *type*, voor zover van toepassing) worden. Vermeld de attributen in de volgorde waarin ze hun waarde krijgen. 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.

---

4. [16 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) Ga er nu vanuit dat we voor het resultaat x van de berekening een register R_x hebben gekozen, voor de argumenten y en z registers R_y en R_z, en dat alle loads en stores zijn uitgevoerd. We kunnen dan daadwerkelijk de volgende assembly instructie uitvoeren:

$$\texttt{ADD}\ R_x, R_y, R_z$$

Hoe moeten de verschillende register descriptors en address descriptors worden aangepast, om het effect van deze assembly instructie te weerspiegelen? Geef de benodigde aanpassingen in een volgorde die efficiënt algoritmisch is uit te voeren.

---

5. [19 pt] Om een array `a` met integers op posities $0 \ldots n-1$ te sorteren, kunnen we gebruik maken van CycleSort. Als we aannemen dat een integer vier bytes in beslag neemt, kan een rechttoe-rechtaan vertaling van een deel van een implementatie van CycleSort naar drie-adres code het volgende opleveren:

```
(1)  pos = k
(2)  i = k+1
(3)  t1 = n-1
(4)  iffalse i <= t1 goto 11
(5)  t2 = 4*i
(6)  t3 = a[t2]
(7)  iffalse t3 < item goto 9
(8)  pos = pos+1
(9)  i = i+1
(10) goto 3
(11) t4 = 4*pos
(12) t5 = a[t4]
(13) iffalse item == t5 goto 16
(14) pos = pos+1
(15) goto 11
(16) t6 = 4*pos
(17) tmp = a[t6]
(18) t7 = 4*pos
(19) a[t7] = item
(20) item = tmp
```

De tijdelijke variabelen in deze code zijn `t1` tot en met `t7`. De variabelen `i` en `tmp` zijn niet *live* na afloop van deze code. De andere niet-tijdelijke variabelen zijn dat wel.

(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) Je moet de drie-adres code hierboven nu stapsgewijs gaan optimaliseren. Bij iedere stap moet je kort aangeven wat je doet, en moet je voor de basic blocks die bij die stap veranderen de complete nieuwe blokken geven. Je mag gebruik maken van de volgende soorten transformaties (voor zover ze te gebruiken zijn):

- *constant folding*
- *local common-subexpression elimination*
- *global common-subexpression eliminiation*
- *copy propagation*
- *dead-code elimination*
- *code motion*
- *reduction in strength*
- *induction-variable elimination*

Noem bij iedere stap het soort transformatie dat je gebruikt hebt. **Stop na maximaal zes optimalisatiestappen.**