Tentamen Vertalerbouw – 9 februari 2007

De nagekeken tentamens zijn af te halen op het onderwijsbureau.

Opmerkingen:

- Schrijf **netjes** en duidelijk, met zwarte of blauwe pen.
- Zet op het eerste blad alle gegevens als naam, etc., en het totaal aantal ingeleverdebladen, en nummer de ingeleverde bladen.
- Lees de opgaven eerst goed door.
- Motiveer uw antwoorden.

1. a) Geef voor alle nonterminals uit onderstaande produkties de sets *first* en *follow*.
   b) Is de grammatica, gegeven door de volgende produkties met startsymbool S, LL(1), LR(0), SLR(1), LR(1)?
   Geef in geval van conflicten deze duidelijk aan. Geef, ingeval de conflicten volgens U oplosbaar zijn, aan hoe de oplossing verloopt.

   S → ABC
   A → aB
   A → b
   B → Bb
   B → S
   C → cA

2. Gegeven is het volgende Pascal-achtige programma:

```
PROGRAM vb;

TYPE rij = array [1..3] of integer;

VAR a: rij;

PROCEDURE set_all (VAR a: rij; b: rij);
VAR i: integer;

   PROCEDURE p (c: rij);
   BEGIN b[i] := c[i]+2;              (* 1 *)
   END;

BEGIN FOR i := 1 TO 3 DO BEGIN
           p(b);                      (* 2 *)
           a[i] := b[i]*2;            (* 3 *)
      END
      ...
END (* set_all *)

BEGIN ...
      set_all(a,a)                    (* 4 *)
      ...
END (* vb *)
```

Voor het geheugenbeheer worden de volgende registers gebruikt:

gp het base address van het activation record van het hoofdprogramma
lnb het base address van het huidige activation record
lfa het adres van de eerste vrije stack locatie

(We gaan ervan uit dat het return adres op een aparte stack wordt bewaard, zodat U daarmee geen rekening hoeft te houden.)

Voor het overdragen van de omgeving van een aan te roepen procedure kan het register **env** worden gebruikt. Verder zijn er voldoende registers (R0, R1, ..) voor het opslaan van tussenresultaten. Merk op dat het keyword `VAR` betekent dat het argument *by reference* wordt doorgegeven.

- Teken het AR (activation record) voor procedure `set_all`;
- Geef de te genereren (pseudo-)instructies voor de **entry** en **exit** van `p`;

Geef de te genereren (pseudo-)instructies voor de vier genummerde regels.

3. De volgende grammatica is topdown parseerbaar.

Prog → *Pdecls Pcall*
Pdecls → *Pdecl Pdecls* | *Empty*
Pdecl → *procsym id lpar Parlst rpar Result semicolon*
Pcall → *callsym id lpar Arglst rpar*
Parlst → *Par Rstpars*
Rstpars → *comma Parlst* | *Empty*
Arglst → *Arg Rstargs*
Rstargs → *comma Arglst* | *Empty*
Par → *intsym* | *charsym*
Result → *Par* | *voidsym*
Arg → *Pcall* | *int* | *character*
Empty →

Een voorbeeld van een zin uit de door deze grammatica beschreven taal is:

```
proc a (int,int) int;
proc b (char,int) void;
call b('c',call a(4,call a(12,17)))
```

Een recursive descent parser bestaat onder andere uit een aantal procedures voor nonterminals van de grammatica.

- Beschrijf in eigen woorden hoe een recursive descent parser te werk gaat.
- Geef de recursive descent code voor de nonterminals Pdecls, Pcall en Arg.
  Hierbij mag je gebruik maken van het volgende:

```
Type tsymbol = ...;
Var sym: tsymbol;
procedure nextsym;
procedure match(fs: tsymbol);
```

- Doe nu hetzelfde als de vorige vraag, maar dan voor een parser met error-recovery. Hierbij mag je gebruik maken van het volgende:

```
Type tsymbol = ...;
Type tsymbolset = Set Of tsymbol;
Var sym: tsymbol;
Procedure nextsym;
Procedure insertion (fs: tsymbol);
Procedure delete (keys: tsymbolset);
Procedure match (fs: tsymbol; keys: tsymbolset);
```

Een mogelijke uitwerking voor het tentamen Vertalerbouw van 9 februari 2007

1a)

	First:	Follow:
S	ab	$abc
A	ab	abc$
B	ab	abc$
C	c	$abc

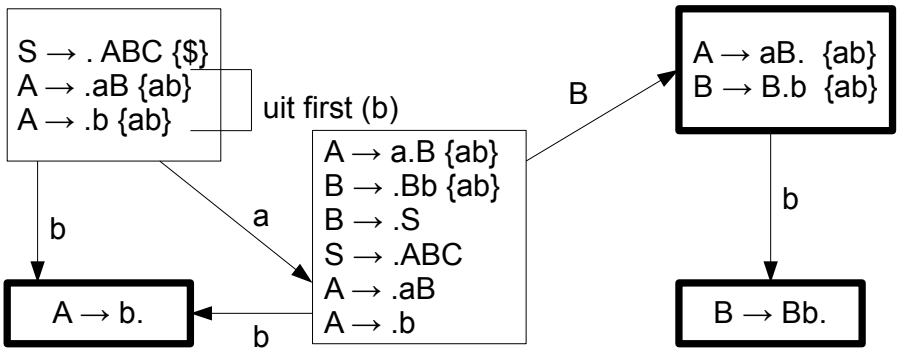
($ staat voor end of file)

1b) De grammatica is niet LL(1), want er is een first/first conflict op beide B → regels.
Poging om het conflict op te lossen:

B → S BH	follow (BH) ⊃ follow (b) ∋ b	
BH → Ɛ	→	first/follow conflict
BH → b BH	first = {b}	

Het is conflict is dus niet makkelijk op te lossen dmv herschrijven.

Nu LR(0) bekijken:



Zie bovenstaand (gedeelte van) state diagram. De grammatica is niet LR(0), want er is een shift-reduce conflict op toestand (A → aB. , B → B . b).
De grammatica is ook niet SLR(1), want b ∈ follow(A), dus nog steeds een shift-reduce conflict.
LR(1): { } / follow sets v. regels toevoegen → nog steeds een shift/reduce conflict.

De conflicten proberen op te lossen heeft geen zin want B =*> Sb* en S =*> α B β, dus er zijn geen geldige parse-trees mogelijk en de taal die beschreven wordt is ÿ

2a) AR van set_all
PN=1, niet genest, dus geen static chain.

	← lfa
i	
old lnb	← lnb
b[3]	←
b[2]	← b by value parameter
b[1]	←
addr a[1]	← a by reference

2b) AR van p
PN=2 (genest), dus static chain erbij

	← lfa
old lnb	
static chain	← lnb (=lfa voor entry)
c[3]	
c[2]	
c[1]	

```
entry:
M(lfa) = env
M(lfa+1) = lnb
lnb = lfa
lfa = lfa + 2

exit:
lfa = lnb
lnb = M(lfa + 1)
return
```

2c1)
```
R0 = M(M(lnb) + 1 )              // i
R1 = M(lnb –  4 + R0)            //  ( [I], -3 addr [i], ondergrens ) -4
R1 = R1 + 2
R2 = M(lnb) – 4 + R0             // addr b[i]
M(R2) = R1
```

2c2)
```
env = lnb
M(lfa) = M(lnb-3)                // push b[1]
M(lfa+1) = M(lnb-2)              // push b[2]
M(lfa+2) = M(lnb-1)              // push b[1]
lfa = lfa + 3
call p
lfa = lfa-3
```

2c3)
```
R0 = M(lnb-4 + M(lnb + 1))       // b[i]
R0 = R0 x 2
R1 = M(lnb-4)                    // addr a, want a by reference
M(R1 – 4 + M(lnb+1)) = R0
```

2c4)
```
M(lfa) = gp                      // push addr a[i]
M(lfa+1) = M(gp)                 // push a[1]
M(lfa+2) = M(gp+1)               // push a[2]
M(lfa+3) = M(gp+2)               // push a[3]
lfa = lfa + 4
call set_all
lfa = lfa – 4
```

3a) Een recursive descent parser is een LL(1) parser die in plaats van een expliciete stack de callstack gebruikt. Voor het matchen van een nonterminal N wordt gekeken naar de first sets van de alternatieven (en mogelijk follow set (N) als $\varepsilon \in$ FIRST (N)) om een alternatief te voorspellen. Als dan bv. $N \rightarrow aAbC$ voorspeld is wordt naar alle (non)terminals gezocht door middel van recursie: match(a); pA(); match(b), pB(); enz.
Hierbij is pX de procedure voor die nonterminal X en match(a) een functie die een 'a' als volgende token van de input leest en deze verwijdert.

3b)
```
procedure Pdecls
begin   if sym ϵ first (Pdecls)
                Pdecl(); pDecls();
        else if sym ϵ follow (Pdecls)
                (* skip *)
        else
                error("Can't match Pdecls");
        end
end

procedure Pcall
begin
        match(callsym); match(d); match(lpar);
        Arglst(); match(rpar);
end

procedure Arg
begin   if sym=int or sym=char
                nextsym
        elseif sym  ϵ first(Arg)
                Pcall();
        else
                error("Can't match Arg");
```

3c)
```
procedure  Pdecls(keys: tsymbolset) (*deze uitwerking lijkt niet helemaal te kloppen*)
begin   delete first(Pdecls);
        if sym  ϵ first (Pdecls)
                Pdecl(keys, first(Pdecls));
                Pdecls(keys)
end

procedure Pcall(keys: tsymbolset)
begin   match (callsym, keys | lpar,rpar,id | first(Arglst))
        match (id, keys | lpar,rpar | first (Arglst))
        match (lpar, keys | rpar | first (Arglst))
        Arglst (keys | rpar))
        match (rpar, keys)
end

procedure Arg(keys: tsymbolset)
begin   delete(keys | first(Arg))
        if sym ϵ first(Pcall) then P call(keys)
        elseif sym= character then nextsym();
        else      //korte keuze voor default productie
        match(int, keys)
        end
end
```

Het tentamen zelf had ik nog liggen, en de uitwerking was een (met de hand geschreven) lastig te lezen PDF document op de FMF site. Ik heb gepoogd e.e.a. te ontcijferen en het resultaat leest u nu.

Met vriendelijke groet,
Dyon Keupink