Løsningsforslag til hjemmeeksamen 2 i INF3110/4110

Roger Antonsen

November 2003

Oppgave 1

Siden oppgaven går ut på å utvide programmet slik at det også genereres kode for prosedyrer, må flere funsjoner endres og legges til. Følgende produksjon i grammatikken skal implementeres; dette gjelder for både (a) og (b).

$\langle program \rangle \rightarrow \langle var\ decl \rangle^? \ \langle proc\ decl\ list \rangle^?$ **begin** $\langle statement\ list \rangle$ **end**

PROGRAM-funksjonen kan endres slik (det som er nytt er merket med rødt).

```
(* ...
   4) saveFreeCode() means: C[2] will contain the instruction 'JUMP
      <main body>', but we don't know yet where this code will be
      located, so we postpone this decision!
   5) Procedures are parsed with PROCDECLLIST
   6) After all procedures are parsed, we are going to parse the main
      body.  We can therefore set the code in C[2]:
      setCode(2,JUMP (CONST (getFreeCode()))
     ... *)

fun PROGRAM ts =
    val (ts1, counter) = (case ts of Var :: tsrest => VARDECL ts
                                   | _ => (ts,0))
    val ts2 = (addCode(setCURRENT(2));
               addCode(setFREE(2+counter));
               saveFreeCode();
               (case ts1 of Procedure :: ts1rest => PROCDECLLIST ts1
                          | _ => ts1))
    val ts3 = parseTerminal Begin ts2
    val ts4 = (setCode(2,JUMP (CONST (getFreeCode())));
               STATEMENTLIST ts3)
    val ts5 = parseTerminal End ts4
in ts5 end
```

a) Løsningsforslag

Implementering av prosedyrer *uten lokale variable.* Dette svarer til følgende variant av ⟨*proc decl*⟩.

⟨*proc decl list*⟩ → ⟨*proc decl*⟩ ⟦; ⟨*proc decl*⟩⟧*
⟨*proc decl*⟩ → **procedure** ⟨*proc name*⟩ **: begin** ⟨*statement list*⟩ **end**

Endringen må hovedsaklig gjøres to steder i koden: **(1)** Først legger vi til `PROCDECLLIST`- of `PROCDECL`-funksjoner, slik at det genereres SIMPLESEM-kode for prosdyredeklarasjoner. **(2)** Deretter endrer vi `STATEMENT` og `IFSTATEMENT` og legger til en `PROCCALL`-funksjon, slik at det også genereres kode for prosedyrekall.

Merk: SIMPLESEM-kode for *retur fra prosedyre* må legges i `PROCDECL`, mens kode for *kall på prosedyre* må legges i `PROCCALL`, jfr. læreboken.

(1) `PROCDECLLIST` blir helt identisk med `PROCDECLLIST` fra parseren for `L2`; denne bare kaller på `PROCDECL` for hver prosedyre som er deklarert.

`PROCDECL` må endres på slik at den passer for produksjonen over. Siden vi ikke har lokale variable, blir dynamiske og statiske lenker unødvendige (se Oppgave 3), og størrelsen på aktiveringsblokken blir lik 1. Det eneste vi da trenger på runtimestakken er returpekeren. Det er derfor ikke nødvendig med *både* `FREE` og `CURRENT` her, men vi har med begge for å gjøre det litt enklere (særlig senere).

I koden under så er det som er nytt i forhold til parseren for `L2` merket med rødt.

```
1  and PROCDECL ts =
2      let val ts1 = parseTerminal Procedure ts
3          val (ts2, name) = PROCNAME ts1
4          val ts3 = parseTerminal Colon ts2
5          val ts4 = (addProcBinding(name,getFreeCode(), 0);
6                     parseTerminal Begin ts3)
7          val ts5 = STATEMENTLIST ts4
8          val ts6 = parseTerminal End ts5
9      in (addCode(SET(CONST 1, CURRENT));
10         addCode(SET(CONST 0, MINUS(CURRENT,CONST 1)));
11         addCode(JUMP(RP));
12         ts6)
13     end
14 and PROCNAME ts =
15     case ts of (Name(name) :: rest) => (rest, name)
16              | _ => raise Compilererror("PROCNAME",ts)
```

Linje 3: `name` blir bundet til en streng som er navnet på prosedyren. `PROCNAME` er en variant av `NAME`, men med en ekstra returverdi, slik at vi får tak i prosedyrenavnet.

Linje 5: En ny prosedyrebinding legges til. `name` er prosedyrenavnet. `getFreeCode()` brukes for å få adressen til første frie kodeplass. (Siden `STATEMENTLIST` i linje 7 vil generere koden som hører til denne prosedyren vil denne adressen være starten på koden som hører til denne prosedyren.) Tallet `0` som sendes som argument brukes aldri; siden det ikke er noen lokale variable tilstede er ikke størrelsen på aktiveringsblokken relevant her.

Linje 9: Ved retur fra prosedyre. Plass blir frigjort ved at FREE settes til CURRENT. (Siden vi vet at størrelse på aktiveringsblokken er lik 1, så svarer dette til å trekke fra 1.)

Linje 10: Ved retur fra prosedyre. CURRENT flyttes til forrige lagerplass, siden størrelsen på aktiveringsblokken er 1.

Linje 11: Ved retur fra prosedyre. Hopp til plassen i *kode*lageret som er lagret i returpekeren (RP).

(2) I STATEMENT og IFSTATEMENT må vi legge til enda et case-tilfelle, slik at også prosedyrekall parses.

```
and STATEMENT ts = case ts of
    (Call :: tsrest)          => PROCCALL ts
  | (Assign :: tsrest)        => ASSIGNMENT ts
  | (Questionmark :: tsrest)  => INPUT ts
  | (Bang :: tsrest)          => OUTPUT ts
  | (If :: tsrest)            => IFSTATEMENT ts
  | _ => raise Compilererror("STATEMENT", ts)
```

PROCCALL tar seg av parseringen av en prosedyrekall. Se i læreboken for alle detaljene her. (Det som er nytt i forhold til parseren for L2 merket med rødt.)

```
1 and PROCCALL ts =
2     let val ts1 = parseTerminal Call ts
3         val (ts2, name) = PROCNAME ts1
4     in (addCode(SET(FREE, CONST(getFreeCode() + 4)));
5         addCode(SET(CONST 0, FREE));
6         addCode(SET(CONST 1, PLUS(FREE, CONST 1)));
7         addCode(JUMP(CONST (getCodeStart(name))));
8         ts2) end
```

Linje 3: name blir bundet til en streng som er navnet på prosedyren som skal kalles.

Linje 4: Returpekeren (offset 0, dvs FREE) settes lik adressen til kodesegmentet som kommer umiddelbart etter dette prosesdyrekallet. getFreeCode() gir oss adressen til første ledige plass i kodelageret; fire plasser senere vil koden som svarer til det som kommer *etter* prosedyrekallet ligge. Det er fire plasser senere fordi vil her legger til fire nye linjer i kodelageret.

Linje 5: CURRENT settes til FREE; dette er egentlig det samme som å legge til 1.

Linje 6: FREE settes til å være FREE pluss 1, siden det ikke er noen dynamiske lenker her. Størrelsen på aktiveringsblokken er kun 1.

Linje 7: Hopp til koden som svarer til prosedyren som heter name. Denne er lagt til som prosedyrebinding ved hjelp av funksjonen addProcBinding (dette gjøres ved parsering av prosedyredeklarasjonen), og adressen til *kodelageret* hentes ved getCodeStart(name).

b) Løsningsforslag

Implementering av prosedyrer *med lokale variable*. Dette svarer til følgende variant av ⟨*proc decl*⟩ (som er den samme som i den opprinnelige grammatikken).

⟨*proc decl*⟩ → **procedure** ⟨*proc name*⟩ **:** ⟨*var decl*⟩$^{?}$ **begin** ⟨*statement list*⟩ **end**

Det er hovedsaklig to endringer som må foretas i forhold til **(a)**. **(1)** `PROCDECL` må endres slik at lokale variable er tillatt. **(2)** Funksjoner for å parse lokale variable må legges til; vi kaller disse `LOCALVARDECL`, `LOCALNAMELIST` og `LOCALNAME`. **(3)** `PROCCALL` må endres slik at det tas hensyn til størrelsen på aktiveringsblokkene.

(1)

```
1  and PROCDECL ts =
2      let val ts1 = parseTerminal Procedure ts
3          val (ts2, name) = PROCNAME ts1
4          val ts3 = parseTerminal Colon ts2
5          val (ts4, counter) =
6                    (case ts3 of Var :: ts3rest => LOCALVARDECL ts3
7                               | _ => (ts3, 0))
8          val ts5 = (addProcBinding(name,getFreeCode(), counter + 2);
9                     parseTerminal Begin ts4)
10         val (ts6) = STATEMENTLIST ts5
11         val ts7 = parseTerminal End ts6
12     in (addCode(SET(CONST 1, CURRENT));
13         addCode(SET(CONST 0, DL));
14         addCode(JUMP(RP));
15         removeVarBindings(counter);
16         ts7) end
```

Linje 5-7: `LOCALVARDECL` returnerer resterende tokens sammen med `counter` som gir antall lokale variable deklarert i denne prosedyren. Hvis det ikke er noen, settes `counter` til 0.

Linje 8: I `addProcBinding` sendes nå som argument størrelsen på aktiveringsblokken til denne prosedyren. Størrelsen er `counter + 2`, siden vi alltid setter av plass til returpeker og dynamisk lenke.

Linje 13: **Ved retur fra prosedyre.** `CURRENT` flyttes til forrige aktiveringsblokk. Adressen til denne fås ved `DL`, den dynamiske lenken, som settes i `PROCCALL`.

Linje 15: Etter at all kode for denne prosedyren er innlest, så fjerner vi de variabelbindingene som er lokale i forhold til denne prosedyren.

(2)

`LOCALVARDECL`, `LOCALNAMELIST` og `LOCALNAME` har *nøyaktig* samme struktur som `VARDECL`, `NAMELIST` og `NAME`. Den eneste relevante forskjellen er i forhold til variabelbindingene. For hver *globale* variabel skjer følgende i `NAME`:

```
addVarBinding(str, CONST (counter + 2));
```

Denne endrer vi nå til følgende i `LOCALNAME`:

```
addVarBinding(str, variableOffset(counter));
```

Eller, ekvivalent:

```
addVarBinding(str, PLUS(CURRENT, CONST (counter+2)));
```

`CURRENT` pluss `counter+2` gir korrekt offset i forhold til den nåværende aktiveringsblokken.

Vi får dermed:

```
and LOCALVARDECL ts =
    let val ts1 = parseTerminal Var ts
        val (ts2, counter) = LOCALNAMELIST (ts1, 0)
        val ts3 = parseTerminal SemiColon ts2
    in (ts3, counter) end
and LOCALNAMELIST (ts, counter) =
    let val (ts1, counter) = LOCALNAME (ts, counter)
    in case ts1 of
            Comma :: ts1rest => LOCALNAMELIST (ts1rest, counter)
          | _ => (ts1, counter) end
and LOCALNAME (ts, counter) =
    case ts of
   (Name(str) :: rest) => (addVarBinding(str, variableOffset(counter));
                           (rest, counter + 1))
  | _ => raise Compilererror("LOCALNAME",ts)
```

(3)

```
1 and PROCCALL ts =
2     let val ts1 = parseTerminal Call ts
3         val (ts2, name) = PROCNAME ts1
4     in (addCode(SET(FREE, CONST(getFreeCode() + 5)));
5         addCode(SET(PLUS(FREE, CONST 1), CURRENT));
6         addCode(SET(CONST 0, FREE));
7         addCode(SET(CONST 1, PLUS(FREE, CONST (getARSize(name)))));
8         addCode(JUMP(CONST (getCodeStart(name))));
9         ts2) end
```

Linje 4: Her settes returpekeren (offset 0, dvs `FREE`). Første ledige plass i kodelageret er nå *fem* plasser etter der hvor koden legges inn. Det er *fem* plasser senere fordi vil her legger til *fem* nye linjer i kodelageret.

Linje 5: Her settes den dynamiske lenken (offset 1, dvs `FREE` pluss 1). Den skal inneholde adressen til det som nå er `CURRENT`.

Linje 7: `FREE` settes til å være `FREE` pluss størrelsen til aktiveringsblokken, som fås ved kallet på `getARSize(name)`.

Oppgave 2

Siden vi ikke har definert LL(1)-kravet for grammatikker i *utvidet* BNF, så er det flere riktige svar på dette spørsmålet.

Kravet til LL(1)-grammatikker (for klassisk BNF) er at de alternative høyresidene for hver produksjon har disjunkte *startmengder*. Hvis noen produksjoner har tomme høyresider, så må vi se på de *utvidede* startmengdene.

(1) La oss først se på de produksjonene som er skrevet uten bruk av utvidet BNF. Ingen av disse produksjonene har en tom høyreside; derfor er det ikke nødvendig å tenke på utvidede startmengder. Vi kan også se bort fra de produksjonene som kun har en høyreside. Da er det er kun tre produksjoner vi trenger å se på:

$\langle \textit{statement} \rangle \to \langle \textit{proc call} \rangle \mid \langle \textit{assignment} \rangle \mid \langle \textit{input} \rangle \mid \langle \textit{output} \rangle \mid \langle \textit{if-statement} \rangle$
$\langle \textit{ar-op} \rangle \to \mathbf{+} \mid \mathbf{-} \mid \mathbf{*} \mid \mathbf{/}$
$\langle \textit{term} \rangle \to \langle \textit{number} \rangle \mid \langle \textit{variable} \rangle$

De alternative høyresidene til ⟨*statement*⟩ gir følgende startmengder: {**call**}, {**assign**}, {**?**}, {**!**} og {**if**}, som er innbyrdes disjunkte.

Høyresidene til ⟨*ar-op*⟩ og ⟨*term*⟩ har åpenbart disjunkte startmengder, siden ⟨*name*⟩ og ⟨*number*⟩ skal betraktes som grunnsymboler.

(2) For hver av produksjonene med utvidet BNF, så er det mulig å analysere situasjonen på flere måter. (Siden LL(1) for utvidet BNF ikke har blitt definert, er man her ganske fri til å gjøre egne antagelser.)

For en produksjon på formen ⟨*A list*⟩ → ⟨*A*⟩ ⟦ ⟨*ny A*⟩ ⟨*A*⟩ ⟧*, så er det rimelig å tolke LL(1)-kravet som følgende: Etter å ha lest én ⟨*A*⟩, så må det være tilstrekkelig å se ett tegn fremover for å avgjøre hvilken produksjon som er den neste. Da er vi nødt til å sammenlikne startmengden til ⟨*ny A*⟩ (i dette tilfellet skal vi lese en ny ⟨*A*⟩) med etterfølgermengden til ⟨*A list*⟩; hvis disse mengdene er disjunkte, så er det gitt hvilket alternativ en skal velge.

For hver produksjon på formen over, så må etterfølgermengden til ⟨*A list*⟩ være disjunkt med startmengden til ⟨*ny A*⟩.

- ⟨*proc decl list*⟩ → ⟨*proc decl*⟩ ⟦ **;** ⟨*proc decl*⟩ ⟧*

Etterfølgermengden til	⟨*proc decl list*⟩	er {**begin**}.
Startmengden til	**;**	er {**;**}.

- ⟨*name list*⟩ → ⟨*name*⟩ ⟦ **,** ⟨*name*⟩⟧*

Etterfølgermengden til	⟨*name list*⟩	er {**;**}.
Startmengden til	**,**	er {**,**}.

- ⟨*statement list*⟩ → ⟨*statement*⟩ ⟦ **;** ⟨*statement*⟩⟧*

Etterfølgermengden til	⟨*statement list*⟩	er {**end**, **fi**}.
Startmengden til	**;**	er {**;**}.

- ⟨*expression*⟩ → ⟨*term*⟩ ⟦ ⟨*ar-op*⟩ ⟨*term*⟩⟧*

Etterfølgermengden til	⟨*expression*⟩	er {**>**, **then**}.
Startmengden til	⟨*ar-op*⟩	er {**+**, **-**, *****, **/**}.

For en produksjon på formen ⟨*A*⟩ → ⟨*B*⟩ $\langle C \rangle^{?}$ ⟨*D*⟩, så er det rimelig å sammenlikne startmengdene til ⟨*C*⟩ og ⟨*D*⟩; hvis disse er disjunkte, så er det mulig å vite, bare ved å se ett tegn fremover, hvorvidt en $\langle C \rangle$ skal leses eller ikke.

Eksempel: I $\langle \textit{proc decl list} \rangle$ forekommer $\langle \textit{var decl} \rangle^?$ **begin**. Her sammenlikner vi start-mengden til $\langle \textit{var decl} \rangle$, som er er lik {**var**}, med mengden {**begin**} og finner at de er disjunkte.

En annen fortolkning her er at produksjonen $\langle A \rangle \rightarrow \langle B \rangle \ \langle C \rangle^? \ \langle D \rangle$ er en "forkortelse" for produksjonene

$$
\begin{aligned}
\langle A \rangle \rightarrow\ & \langle B \rangle \ \langle C \rangle \ \langle D \rangle \ | \\
& \langle B \rangle \ \langle D \rangle
\end{aligned}
$$

Og i denne oversettelsen er LL(1)-kravet *ikke* oppfylt.

(I og med at LL(1) for utvidet BNF ikke har blitt definert i dette kurset, så godtas dette svaret også, hvis det er tilstrekkelig godt begrunnet.)

Oppgave 3

a)

Løsningsforslag

(1) Siden lokale variable ikke forekommer, så blir størrelsen på hver aktiverings-blokk konstant. Å finne frem til forrige prosedyres aktiveringsblokk (det vi ellers bruker dynamiske lenker til) er her veldig enkelt. Uten dynamiske lenker, så blir størrelsen på hver aktiveringsblokk lik én, og forrige aktiveringsblokk finner vi ved å gå en plass opp i datalageret. Så, dynamisk lenker er *ikke* nødvendig her.

(2) Statiske lenker er ikke nødvendig her, siden nøsting av blokker ikke forekommer. Skopet til en variabel er avgrenset av kroppen til prosedyren.

b)

Løsningsforslag

(1) Siden lokale variable forekommer her, så får aktiveringsblokkene ulike størrelser. Dynamiske lenker brukes for å kunne peke til begynnelsen på forrige aktiverings-blokk.

(2) Av samme grunn som over er ikke statiske lenker nødvendige her heller.

Oppgave 4

I denne oppgaven **SM** for startmengde, **EM** for etterfølgermengde og **USM** for utvidet startmengde.

a)

Løsningsforslag

Den er ikke LL(1). Begrunnelse: startmengdene er ikke disjunkte.

Produksjon	SM
$\langle A\rangle \to$ **a** $\langle A\rangle\ \langle F\rangle$	a
$\to$ **a** $\langle B\rangle\ \langle F\rangle$	a
$\to$ **a**	a
$\langle B\rangle \to \langle B\rangle$ **b**	c
$\to$ **c**	c
$\langle F\rangle \to$ **f**	f

En ekvivalent LL(1)-grammatikk:

$\langle A\rangle \to$ **a** $\langle Ax\rangle$
$\langle Ax\rangle \to \langle A\rangle\langle F\rangle \mid \langle B\rangle\ \langle F\rangle \mid \varepsilon$
$\langle B\rangle \to$ **c** $\langle C\rangle$
$\langle C\rangle \to$ **b** $\langle C\rangle \mid \varepsilon$
$\langle F\rangle \to$ **f**

Begrunnelse for at den er LL(1): de utvidede startmengdene er disjunkte.

Produksjon	SM	EM	USM
$\langle A\rangle \to$ **a** $\langle Ax\rangle$	a	f	a
$\langle Ax\rangle \to \langle A\rangle\langle F\rangle$	a	f	a
$\to \langle B\rangle\langle F\rangle$	c		c
$\to \varepsilon$	-		f
$\langle B\rangle \to$ **c** $\langle C\rangle$	c	f	c
$\langle C\rangle \to$ **b** $\langle C\rangle$	b	f	b
$\to \varepsilon$	-		f
$\langle F\rangle \to$ **f**	f	f	f

b)

Løsningsforslag

Den er ikke LL(1). Begrunnelse: startmengdene er ikke disjunkte.

Produksjon	SM
$\langle S\rangle \to \langle A\rangle$ **a**	b
$\to$ **b**	b
$\langle A\rangle \to \langle S\rangle\ \langle B\rangle$	b
$\langle B\rangle \to$ **ab**	a

For å finne en ekvivalent LL(1)-grammatikk er det lurt å se hva denne grammatikken egentlig uttrykker. Enhver $\langle A\rangle$ gir $\langle S\rangle\langle B\rangle$ og enhver $\langle B\rangle$ gir **ab**; det betyr at enhver $\langle A\rangle$ gir $\langle S\rangle$**ab**. Fra den første produksjonen får vi dermed at $\langle S\rangle$ gir $\langle S\rangle$**aba** eller **b**. Hele grammatikken kan dermed skrives slik (og det som gjenstår er å fjerne venstrerekursjonen):

$\langle S \rangle \to \langle S \rangle$ **aba** | **b**

En ekvivalent LL(1)-grammatikk:

$\langle S \rangle \quad \to$ **b** $\langle SX \rangle$
$\langle SX \rangle \to$ **aba** $\langle SX \rangle \mid \varepsilon$

Begrunnelse for at den er LL(1): de utvidede startmengdene er disjunkte.

Produksjon	SM	EM	USM
$\langle S \rangle \quad \to$ **b** $\langle SX \rangle$	**b**	-	**b**
$\langle SX \rangle \to$ **aba** $\langle SX \rangle$	**a**	-	**a**
$\quad \to \varepsilon$	-		-

c)

Løsningsforslag

Den er ikke LL(1). Begrunnelse: startmengdene er ikke disjunkte.

Produksjon		SM
$\langle T \rangle \to$	$\langle T \rangle \vee \langle F \rangle$	¬ **(true false**
$\to$	$\langle F \rangle$	¬ **(true false**
$\langle F \rangle \to$	$\langle F \rangle \wedge \langle S \rangle$	¬ **(true false**
$\to$	$\langle S \rangle$	¬ **(true false**
$\langle S \rangle \to$	$\neg \langle P \rangle$	¬
$\to$	$\langle P \rangle$	**(true false**
$\langle P \rangle \to$	**(** $\langle T \rangle$ **)**	**(**
$\to$	**true**	**true**
$\to$	**false**	**false**

En ekvivalent LL(1)-grammatikk:

$\langle T \rangle \quad \to \langle F \rangle \langle Fs \rangle$
$\langle Fs \rangle \to \vee \langle F \rangle \langle Fs \rangle \mid \varepsilon$
$\langle F \rangle \quad \to \langle S \rangle \langle Ss \rangle$
$\langle Ss \rangle \to \wedge \langle S \rangle \langle Ss \rangle \mid \varepsilon$
$\langle S \rangle \quad \to \neg \langle P \rangle \mid \langle P \rangle$
$\langle P \rangle \quad \to$ **(** $\langle T \rangle$ **)** | **true** | **false**

Begrunnelse for at den er LL(1): de utvidede startmengdene er disjunkte.

Produksjon	SM	EM	USM
$\langle T \rangle \to \langle F \rangle \langle Fs \rangle$	¬ **(true false**	**)**	¬ **(true false**
$\langle Fs \rangle \to \vee \langle F \rangle \langle Fs \rangle$	$\vee$	**)**	$\vee$
$\quad \to \varepsilon$	-		**)**
$\langle F \rangle \to \langle S \rangle \langle Ss \rangle$	¬ **(true false**	**)** $\vee$	¬ **(true false**
$\langle Ss \rangle \to \wedge \langle S \rangle \langle Ss \rangle$	$\wedge$	**)** $\vee$	$\wedge$
$\quad \to \varepsilon$	-		**)** $\vee$
$\langle S \rangle \to \neg \langle P \rangle$	¬	**)** $\vee$ $\wedge$	¬
$\quad \to \langle P \rangle$	**(true false**		**(true false**
$\langle P \rangle \to$ **(** $\langle T \rangle$ **)**	**(**	**)** $\vee$ $\wedge$	**(**
$\quad \to$ **true**	**true**		**true**
$\quad \to$ **false**	**false**		**false**

Oppgave 5

a) Løsningsforslag

Nei, LL(1)-grammatikker kan ikke være flertydige. En skisse til bevis (ved motsigelse) går som følger. Anta at det fins en LL(1)-grammatikk som *er* flertydig. Da må det finnes en streng i det tilhørende språket som har to ulike syntakstrær. Se på hvordan disse syntakstrærne kan konstrueres fra en LL(1)(recursive descent)-parser. For hvert metasymbol og hvert grunnsymbol skal det finnes nøyaktig én tilhørende produksjon i grammatikken. Men, siden det er to syntakstrær for denne strengen, vil det finnes en situasjon hvor det er mulig å velge mellom to ulike produksjoner. Da er ikke grammatikken LL(1), og det gir en motsigelse. Det kan altså ikke finnes en LL(1)-grammatikk som er flertydig.

b) Løsningsforslag

En venstrerekursiv grammatikk er nødt til å ha en produksjon på følgende form:

$$\langle P \rangle \rightarrow \langle P \rangle\ \langle Q \rangle \mid \langle R \rangle$$

Startmengden til $\langle P \rangle$ er her lik startmengden til $\langle R \rangle$. Dvs. at de to høyresidene har nøyaktig like startmengder; dermed er ikke grammatikken LL(1).