# Oppgaver til INF 5110, kapittel 4, med svarforslag

Gjennomgås tirsdag 21. febr. 2012

Oppgave 1: Sjekk om grammatikken ”S → ( S ) S | ε” er LL(1)

Oppgave 2: Gitt gram.: exp → exp **+** exp | **(**exp**)** | **if** exp **then** exp **else** exp | **var**

a) Lag en entydig grammatikk for dette språket, der + skal være venstreassosiativ, og der ”if x then y else z+u” skal bety ”if x then y else (z+u)”.

b) Hvorfor får vi ikke noe ”dangling else”-problem her?

Oppgave 3 (Mye repetisjon. Blir ikke fullt gjennomgått, man fullt svarforslag gis på foiler):

Gitt gram.: exp → exp op exp | (exp) | **num**
op → + | - | * | / | ** | < | =

a) Grammatikken over er opplagt flertydig. Lag en *entydig* grammatikk for språket ut fra at følgende tilleggsregler:
   - ** (opphøying) har presedens 3 (høyest) og er høyre-assossiativ
   - * og / har presedens 2, og er venstre-assosiaiv
   - + og - har presedens 1 og er venstre-assosiativ
   - < og = har presedens 0, og er ikke-assosiativ

b) Se på grammatikken du fant under a), og skriv et syntaksdiagram (med løkker der det passer) for hver ikke-terminal. Del opp ”op”- terminalene på hensiktsmessig måte.

c) Lag recursive-descent prosedyrer for å sjekke programmet (med while-setninger der det passer) ut fra grammatikken fra b). Du kan bruke både ”match(token)” og ”getToken()” fra boka (som begge setter neste symbol inn i variablen ”token”).

d) Ut fra svaret på c), legg til trebyggings-setninger i prosedyren som behandler en sekvens av ** slik at treet får riktig høyre-assosiativ form.

e) Ta hele grammatikken fra a), og gjør den fri for venstreassosiativitet, og gjør all mulig venstrefaktorisering (men behold entydighet).

f) Sjekk om grammatikken fra e) er LL(1).

# Oppgave 1

Sjekk om grammatikken ”S → ( S ) S | ε” er LL(1)

| | First | Follow |
|---|---|---|
| S | ε ( | $ ) |

Tabellen for valg av alternativ blir dermed:

| | ( | ) | $ |
|---|---|---|---|
| S | S →( S ) S | S → ε | S → ε |

Og denne tabellen er entydig, altså er den LL(1).

En recursive descent prosedyre kunne bli (ikke spurt om i oppgaven):

```
procedure S() {
    if token = ”(” then {
        getToken();
        S();
        match( ”)” );
        S();
    } else {
        // ingen ting
    }
}
```

## For interesserte:

Grammatikken: S → S ( S ) | ε (som gir samme språk) er derimot *ikke* LL(1). Vi får her:

| | First | Follow |
|---|---|---|
| S | ε ( | $ ( ) |

Dermed blir det konflikt for ”(”

**DESSUTEN er den altså venstrerekursiv,** så vi kunne egentlig umiddelbart sagt at den ikke er LL(1)!

# Oppgave 2

**Oppgave:** Gitt gram.: exp → exp **+** exp | **(**exp**)** | **if** exp **then** exp **else** exp | **var**

a) Lag en entydig grammatikk for dette språket, der + skal være venstreassosiativ, og der "if x then y else z+u" skal bety "if x then y else (z+u)".

exp → exp + exp1 | exp1
exp1 → if exp then exp else exp | ( exp ) | **var**

Denne *er* entydig (den er SLR(1), som vi kommer til). Merk at vi f.eks. kan ha setningen:
a + b + if c then d else e + f. Denne vil bli tolket slik:
(a + b) + (if c then d else (e + f)).
Setningen:
(a + b) + (if c then d else (if g then h else (e + f))) får den betydningen som angitt om den skrives helt uten parenteser.

b) Hvorfor får vi ikke noe "dangling else"-problem her?

Det kommer av at det ikke er noe tvil om det skal være med en *else* eller ikke til en *if-then*. Det skal *alltid* være med en else!

# Oppgave 3a

Gitt grammatikken: exp → exp op exp | (exp) | **num**
op → + | - | * | / | ** | < | =

a) Grammatikken over er opplagt flertydig. Lag en entydig grammatikk for språket ut fra at følgende tilleggsregler:

** (opphøying) har presedens 3 (høyest) og er høyre-assossiativ
* og / har presedens 2, og er venstre-assosiaiv
+ og - har presedens 1 og er venstre-assosiativ
< og = har presedens 0, og er ikke-assosiativ

## Svarforslag:

Vi må lage en ny ikke-terminal for hvert presedens-nivå. Vi velger fra laveste til høyeste:

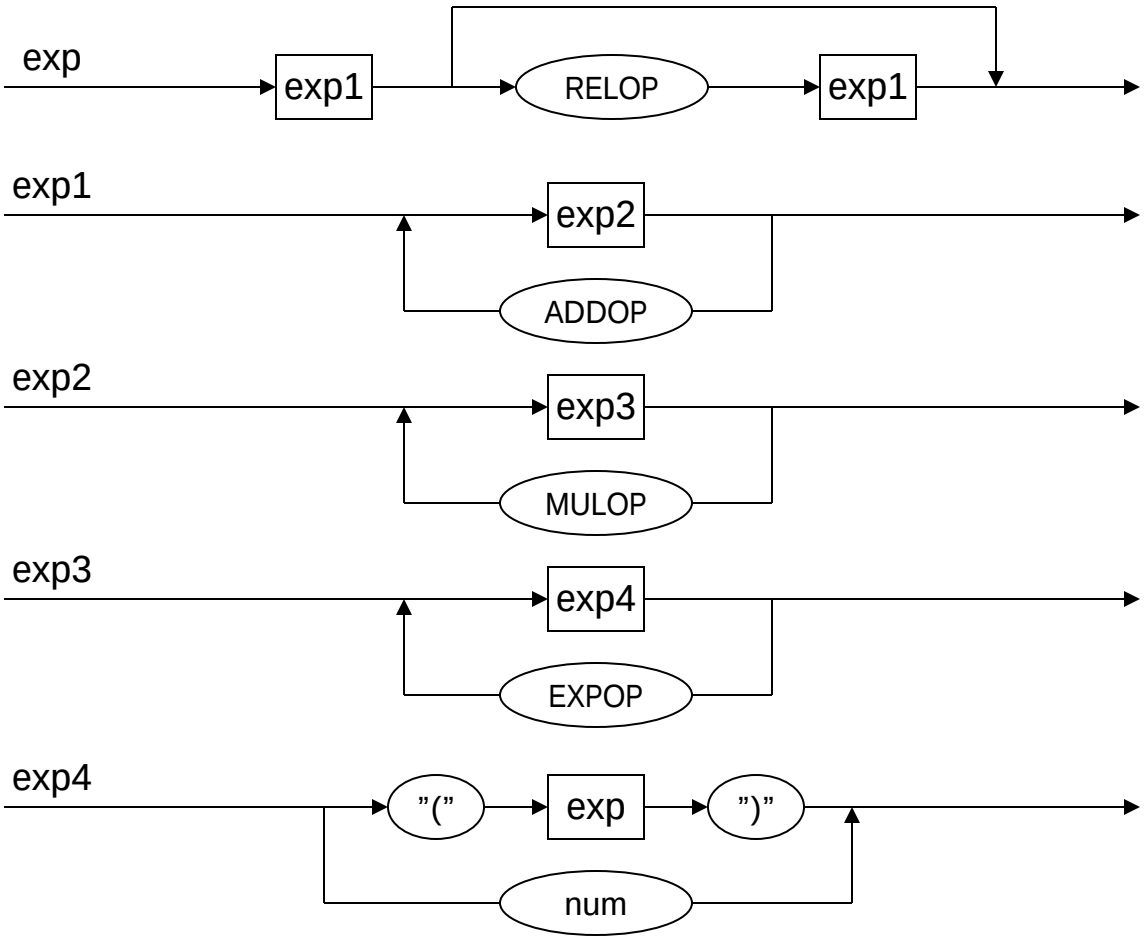
| | | |
|---|---|---|
| exp | som den er i oppgaven | |
| exp1 | for operander foran og bak < og = | |
| exp2 | for operander mellom, foran og bak + og – | (term) |
| exp3 | for operander mellom, foran og bak * og / | (faktor) |
| exp4 | for operander mellom, foran og bak ** | (opphøying) |

**Grammatikken blir:**

| | |
|---|---|
| exp → exp1 RELOP exp1 \| exp1 | RELOP dekker < og =, men kommer som samme token |
| exp1 → exp1 ADDOP exp2 \| exp2 | Tilsvarende for + og - |
| exp2 → exp2 MULOP exp3 \| exp3 | Tilsvarende for * og / |
| exp3 → exp4 EXPOP exp3 \| exp4 | Tilsvarende, men snudd, for ** |
| exp4 → ( exp ) \| **num** | |

# Oppgave 3b

**Oppgaven:** Se på grammatikken du fant under a), og skriv et syntaksdiagram (med løkker der det passer) for hver ikke-terminal. Del opp ”op”- terminalene på hensiktsmessig måte.




**Merk:**
Assosiativitet kommer ikke fram her. Det må eventuelt legges inn i trebyggingen i spørsmål d)

# Oppgave 3c

c) **Oppgaven:** Lag recursive-descent prosedyrer for å sjekke programmet (med while-setninger der det passer) ut fra grammatikken fra b). Du kan bruke både "match(token)" og "getToken()" fra boka (som begge setter neste symbol inn i variablen "token").

```
procedure exp( ) {
    exp1( );
    if token = RELOP then {
        getToken( )
        exp1( );
    }
}
```

```
procedure exp1( ) {
    exp2;
    while token = ADDOP do {
        getToken( );
        exp2( );
    }
}
```

Både exp2 og exp3 blir helt tilsvarende til exp1( )

```
procedure exp4( ) {
    if token = LPAR then {
        getToken( );
        expr( );
        match( RPAR )
    } else {
        match( NUM );
    }
}
```

Om "exp" er det faktisk **ytterste** startsymbolet (som ofte heter "program"), så legger man gjerne på en ytterste rec.descent-prosedyre som får det hele riktig i gang, og som sjekker at det ikke er noe grums **etter** programmet. Den kan f.eks. være slik:

```
procedure expression( ) {
    getToken( );
    exp( );
    if token != "$" then {error("grums etter
                                 uttrykket");}
}
```

# Oppgave 3d (se alternativ løsning på neste foil)

NB: Det var noe feil på denne foilen i det som er delt ut tidligere! Er rettet her og i de endelige foiler som legges ut.

**Oppgaven:** Ut fra svaret på c), legg til trebyggings-setninger i prosedyren som behandler en sekvens av **, slik at treet får riktig høyre-assosiativ form.

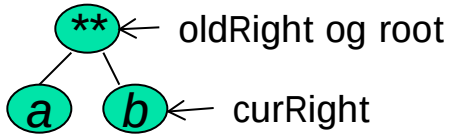
*a ** b ** c*

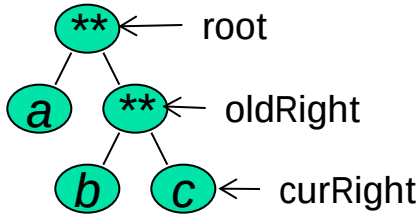
```
procedure exp3(): TreeNode {
   TreeNode  oldRight, curRight, newRight,  newOp;

   curRight = exp4();  root = curRight;  oldRight = null;
   while token = EXPOP do {
      getToken();
      newRight = exp4();
      newOp = new OpNode("**");  newOp.right = newRight;
      if  (oldRight == null) {
         newOp.left= curRight ; root = newOp;
      } else {
         newOp.left = oldRight.right ; oldRight.right = newOp;
      }
      oldRight = newOp; curRight = newRight;
   }
   return root;
}
```

oldRight == null

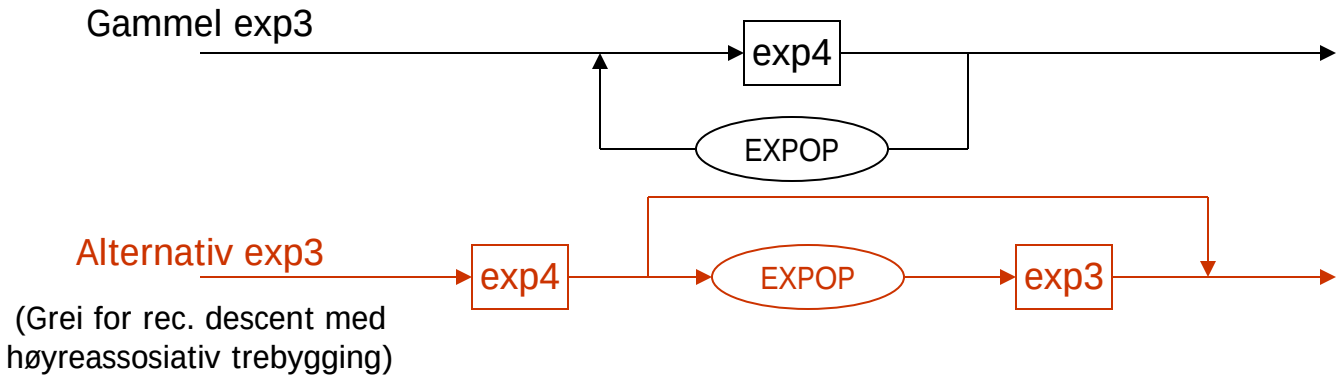
*Før* while-setn: a ← curRight og root




newRight og newOP er alltid ute av bruk mellom iterasjonene

# Oppgave 3d alternativ løsning




(Grei for rec. descent med høyreassosiativ trebygging)

Over er angitt et alternativt syntaksdiagram for exp3. Dette kan gi en rec. desc. prosedyre, med greiere høyreassoiativ trebygging. Det kan være slik:

**Uten trebygging:**

```
procedure exp3() {
    exp4;
    if token = EXPOP then {
        getToken();
        exp3();
    }
}
```

**Med trebygging:**

```
procedure exp3(): TreeNode {
    TreeNode root; OpNode opNode;
    root = exp(4);
    if token = EXPOP then {
      getToken();
      newRight = exp3();
      root = new OpNode("**", root,
                        newRight);
    }
    return root;
}
```

# Oppgave 3e

e) Ta hele grammatikken fra a), og gjør den fri for venstreassosiativitet, og gjør all mulig venstrefaktorisering (men behold entydighet).

exp → exp1 RELOP exp1 | exp1 — RELOP dekker < og =, men er samme token
exp1 → exp1 ADDOP exp2 | exp2 — Tilsvarende for + og -
exp2 → exp2 MULOP exp3 | exp3 — Tilsvarende for * og /
exp3 → exp4 EXPOP exp3 | exp4 — Tilsvarende for **
exp4 → ( exp ) | **num**

exp → exp1 expx
expx → RELOP exp1 | ε
exp1 → exp2 exp1x
exp1x → ADDOP exp2 exp1x | ε
exp2 → exp3 exp2x
exp2x → MULOP exp3 exp2x | ε
exp3 → exp4 exp3x
exp3x → EXPOP exp3 | ε
exp4 → ( exp ) | **num**

At vi faktisk beholder entydighet er ikke uten videre greit å se, men det blir klart i oppgave **f)**, siden vi der finner at grammatikken er LL(1). Alle grammatikker som er LL(1) er entydige.

# Oppgave 3f

f) Sjekk om grammatikken fra e) er LL(1). Vi beregner først First og Follow (Fi og Fo). FiU er Fi uten ε. Regner her RELOP, ADOP, MULOP og EXPOP som teminal-symboler. Vi gjentar de røde aksjonene til det stabiliserer seg.

| | |
|---|---|
| exp → exp1 expx | Legger Fo(exp) inn i Fo(expx) og inn i Fo(exp1). Legger FiU(expx) inn i Fo(exp1) |
| expx → RELOP exp1 \| ε | Legger Fo(expx) inn i Fo(exp1). Legger RELOP inn i Fi(expx) |
| exp1 → exp2 exp1x | Legger Fo(exp1) inn i Fo(exp1x) og inn i Fo(exp2). Legger FiU(exp1x) inn i Fo(exp2) |
| exp1x → ADDOP exp2 exp1x \| ε | Legger Fo(exp1x) inn i Fo(exp2). Legger FiU(exp1x) inn i Fo(exp2). Legger ADDOP inn i Fi(exp1x) |
| exp2 → exp3 exp2x | Legger Fo(exp2) inn i Fo(exp2x) og inn i Fo(exp3). Legger FiU(exp2x) inn i Fo(exp3) |
| exp2x → MULOP exp3 exp2x \| ε | Legger Fo(exp2x) inn i Fo(exp3). Legger FiU(exp2x) inn i Fo(exp3) Legger MULOP inn i Fi(exp2x) |
| exp3 → exp4 exp3x | Legger Fo(exp3) inn i Fo(exp3x) og inn i Fo(exp4). Legger FiU(exp3x) inn i Fo(exp4) |
| exp3x → EXPOP exp3 \| ε | Legger Fo(exp3x) inn i Fo(exp3). Legger EXPOP inn i Fi(exp3x) |
| exp4 → ( exp ) \| **num** | Legger ")" inn i Fo(exp). Legger "(" og **num** inn i Fi(exp4) |

| | **First** | **Follow** | |
|---|---|---|---|
| exp | num ( | $ ) | Legger først $ inn i Follow(exp) (siden exp er startsymbolet) |
| expx | ε RELOP | $ ) | |
| exp1 | num ( | $ ) RELOP | |
| exp1x | ε ADDOP | $ ) RELOP | |
| exp2 | num ( | $ ) RELOP ADDOP | |
| exp2x | ε MULOP | $ ) RELOP ADDOP | |
| exp3 | num ( | $ ) RELOP ADDOP MULOP | |
| Exp3x | ε EXPOP | $ ) RELOP ADDOP MULOP | |
| exp4 | num ( | $ ) RELOP MULOP ADDOP EXPOP | |

| | **RELOP** | **ADDOP** | **MULOP** | **EXPOP** | **num** | **(** | **)** | **$** |
|---|---|---|---|---|---|---|---|---|
| exp | | | | | exp1 expx | exp1 expx | | |
| expx | RELOP exp1 | | | | | | ε | ε |
| exp1 | | | | | exp2 exp1x | exp2 exp1x | | |
| exp1x | ε | ADDOP exp2 exp1x | | | | | ε | ε |
| exp2 | | | | | exp3 exp2x | exp3 exp2x | | |
| exp2x | ε | ε | MULOP exp3 exp2x | | | | ε | ε |
| exp3 | | | | | exp4 exp3x | exp4 exp3x | | |
| exp3x | ε | ε | ε | EXPOP exp3 | | | ε | ε |
| exp4 | | | | | num | ( exp ) | | |

Dermed, siden det ikke er konflikter: Den er LL(1)! Og videre: Dermed også entydig.