Oppgaver til INF 5110, kapittel 4, med svarforslag

Gjennomgått torsdag 14. febr. 2008. Disse foilene er justert 15/2, kl. 11

Oppgave 1 (Mye repetisjon): 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).

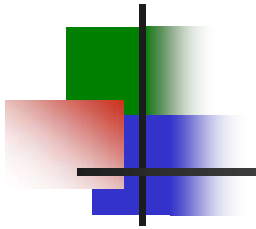
Oppgave 2: Skriv om prosedyrene midt på side 162, slik at de produserer trær (og ikke tall)

Oppgave 3: Sjekk om grammatikken "S → (S) S | ε" er LL(1)

Oppgave 4: 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)". Forsøk også å lage en grammatikk med den "motsatte" tolkningen: at det betyr "(if x then y else z)+u".

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



Oppgave 1

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:

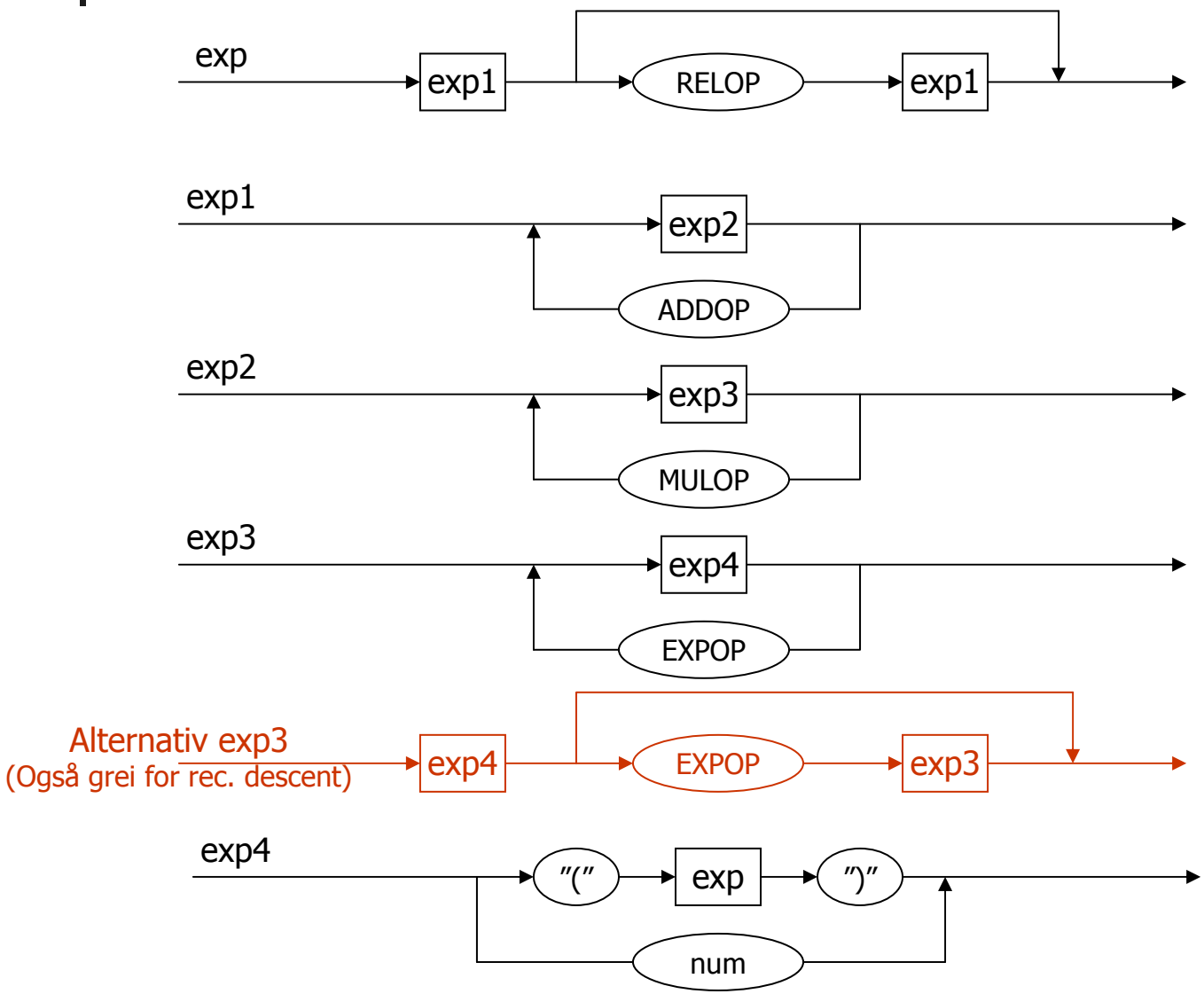
exp	som den er i oppgaven	
exp1	for operander foran og bak < og =	
exp2	for operander mellom, foran og bak + og –	(term)
exp3	for operander mellom, foran og bak * og /	(faktor)
exp4	for operander mellom, foran og bak **	(opphøying)

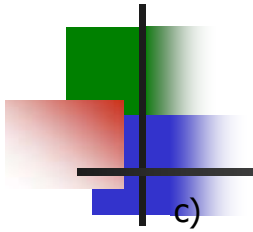
Grammatikken blir:

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 1

Svarforslag til b)





Oppgave 1

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”).

```
exp1(); procedure exp() {
    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 );
    }
}
```

Oppgave 1

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.

```
procedure exp3(): TreeNode {
   Treenode root, newRight;        OpNode opNode, oldOpNode;
   root = exp4();
   if token = EXPOP then {
       getToken();
       newRight = exp4();
       opNode = new OpNode("**", root, newRight);
       root = opNode;
   }
   while token = EXPOP do {
       getToken();
       newRight = exp4();
       oldOpNode = opNode;
       opNode = new OpNode("**", oldOpNode.right , newRight);
       oldOpNode.right = opNode;
   }
   return root;
}
```

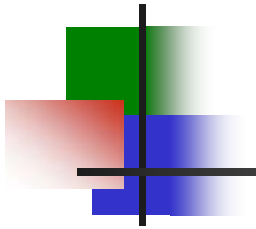
Lagt til etter gjennomgåelsen:

Om exp er det faktisk ytterste startsymbolet, 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");}
}
```

Men merk at man godt kan bruke den alternative syntaksen (den i rødt, som ikke er venstre-rekursiv) for denne operatoren. Da får man enklere prosedyre, omtrent slik:

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



Oppgave 1

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**

Lagt til etter gjennomgåelsen:

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 1

Det i rødt er lagt til etter gjennomgåelsen

f) Sjekk om grammatikken fra e) er LL(1). Vi beregner først First og Follow (Fi og Fo). FiU er Fi uten ε

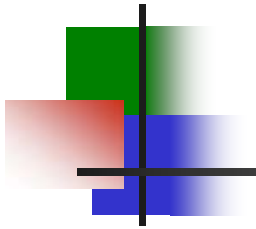
exp → exp1 expx	Legger Fo(exp) inn i Fo(expx) og inn i Fo(exp1). Legger FiU(expx) inn i Fo(exp1)
expx → RELOP exp1 | ε	Legger Fo(expx) inn i Fo(exp1)
exp1 → exp2 exp1x	Legger Fo(exp1) inn i Fo(exp1x) og inn i Fo(exp2). Legger FiU(exp1x) inn i Fo(exp2)
exp1x → ADDOP exp2 exp1x | ε	Legger Fo(exp1x) inn i Fo(exp2). Legger FiU(exp1x) inn i Fo(exp2)
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(exp1x) inn i Fo(exp3)
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).
exp4 → (exp) | **num**	Legger ")" inn i Fo(exp)

	First	**Follow**
exp	num (	$)
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

Legger først $ inn i Follow(exp) (siden exp er startsymbolet)

	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 altså: Dermed også entydig.



Oppgave 2

Skriv om prosedyrene midt på side 162, slik at de produserer trær (og ikke tall). Dette er på en måte det omvendte av det i oppgave 1d.

```
function exp(): TreeNode;
   var temp: TreeNode;
begin
       temp = term();
       return exp'(temp);
end;

function exp'(treelSoFar: TreeNode)
begin
    if token ==  "+" or token == "-"  then
       case token of
          +: match("+");
             treeSoFar = new OpNode("+", treeSoFar, term());
          - : match("-");
             treeSoFar = new OpNode("-", treeSoFar, term());
       end case;
       return exp'(treeSoFar);
    else
       return treeSoFar;
    end;
end
```

Oppgave 3

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).

For interesserte:

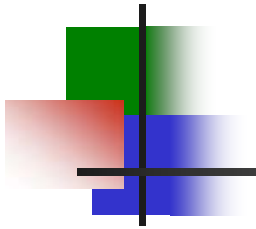
Grammatikken: S → S (S) | ε
(som gir samme språk) er derimot *ikke* LL(1). Vi får her:

	First	Follow
S	ε (	$ ( )

Dermed blir det konflikt for "("

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

```
procedure S() {
    if token = "(" then {
        getToken();
        S();
        match( ")" );
        S();                    // Lagt til etter gjennomgåelsen
    } else {
        // ingen ting
    }
}
```



Oppgave 4

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)". Forsøk også å lage en grammatikk med den "motsatte" tolkningen: at det betyr "(if x then y else z)+u".

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

Etter gjenomgåelsen: Denne *er* entydig (den er SLR(1)). 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. Grammatikken under er også entydig, og gjør det den skal.

Den motsatte grammatikken gir et veldig rart språk, siden vi kan bruke + inni if- og else-uttrykket, men ikke inni then-uttrykket. Men følgende grammatikk vill gjøre susen:

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

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!