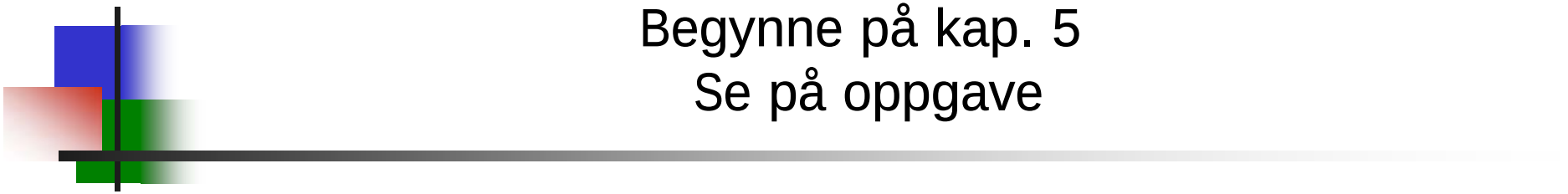


INF5110, 15/2-2007

Dagens temaer:

Avslutning kap. 4

Begynne på kap. 5

Se på oppgave

Stein Krogdahl,

Ifi UiO

Avslutning kap. 4

1. Vi tar ikke med i pensum noe om trebygging ut fra LL(1)-parsering med stakk (altså 4.2.4).
   - Vi ser på denne type parsering bare som en forberedelse til LR-parsering, der vi har en tilsvarende
2. Man kan snakke om lengere ”lookahead” enn 1 for LL-grammatikker, og får da LL(k)-grammatikker.
   - Det teoretiske rundt dette er litt komplisert, så det skal vi ikke se på (se kap 4.3.4)
   - Derimot er det viktig å vite at mange LL-verktøy tillater at du i bestemte situasjoner titter et antall symboler framover for å få en avgjørelse på hva slags konstruksjon du går inn i.
   - F.eks. om symbolet ”static” kommer inne i en klasse, så må vi lese noen symboler fremover få å finne ut om dette er variabel-dekl. eller en metode-dekl.

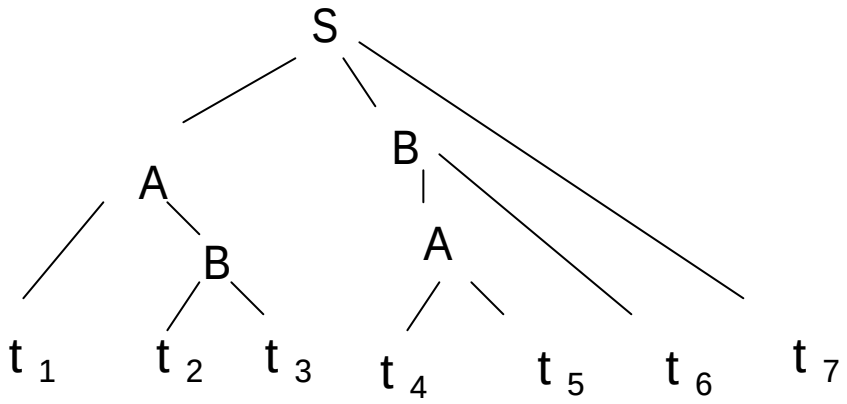
Om igjen: Litt om stoffet i kap. 4

Dette bør leses om igjen etter kapittelet:

- First og Follow-mengder
  - Boka tar det et stykke uti, vi tok det først
- Begrepet LL(1)-parsering (kommer)
  - Boka reserverer dette begrepet for den metoden (kap 4.2 s. 152) der man bruker en eksplisitt stakk (isteden for rekursive metoder, som i ”recursive decent”-framgangsmåten)
  - Det vanlige er å bruke betegnelsen LL(1)-parsering også når rec.decent metoden brukes slavisk på en ren BNF-grammatikk
  - Kravet til en LL(1)-grammatikk kommer like tydelig fram ut fra begge disse metodene. Vi skal tenke like mye rec.decent som eksplisitt stakk. (boka gjør bare det siste).
  - Ofte brukes betegnelsen LL(1)-parsering også om rec.decent metoden brukt ut fra syntaksdiagrammer eller EBNF, men da er det uklart hva LL(1)-kravet til en grammatikk er.

Start på kapittel 5:

”Bottom up” parsering (nedenfra-og-opp)



LR-parsering og grammatikker

- LR(0)
- SLR(1)
- LR(1)
- LALR(1)

-Autimatisert

-YACC, Bison (LALR(1))

Prinsippet og datastrukturen for LR-parsering

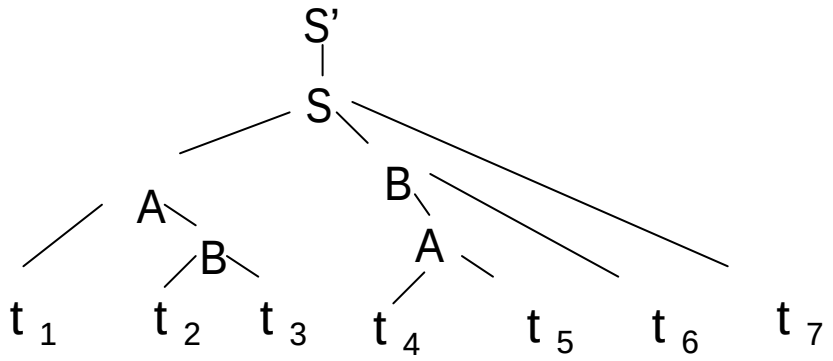
$S' \rightarrow S$

$S \rightarrow A\ B\ t_7 \mid ....$

$A \rightarrow t_4\ t_5 \mid t_1\ B \mid$

$B \rightarrow t_2\ t_3 \mid A\ t_6 \mid ....$

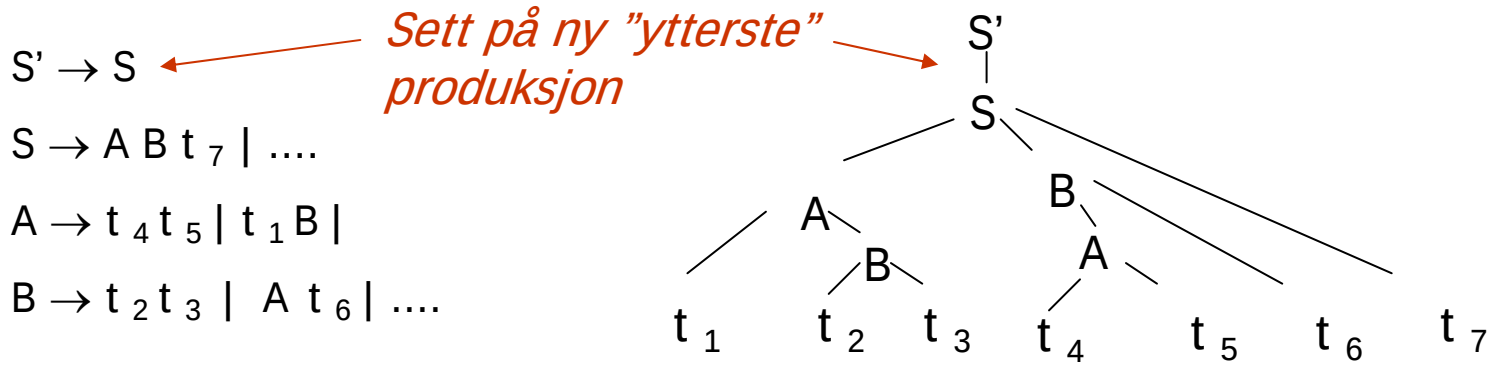
Anta at grammatikken er entydig, og at vi *kjenner syntaks-treet* for setningen:



LR-parsering :

- Ha en ”stakk” for *det som er lest* (altså *omvendt* av LL-parsering med stakk)
- Gjør ”reduksjonen” av et subtre når det ligger ”på toppen av” stakken

Prinsippet og datastrukturen LR-parsering II



Anta at grammatikken er entydig, og anta at vi kjenner syntaks-treet for setningen:

- Ha en "stakk" for det som er lest
- Gjør "reduksjonen av et subtre når subtreet ligger "på toppen av" stakken.
- Da erstatter vi det på stakken med den ikke-terminalen (som produserte dette subtreet)

Start-situasjonen:

$ t_1 t_2 t_3 t_4 t_5 t_6 t_7 $

stakk input

Slutt-situasjonen:

$ S' $

stakk

$

input

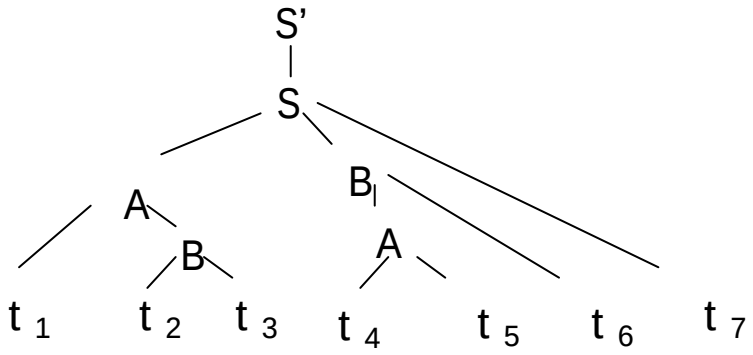
Prinsippet med LR-parsering III

S' → S

S → A B t_7 | ....

A → t_4 t_5 | t_1 B |

B → t_2 t_3 | A t_6 | ....



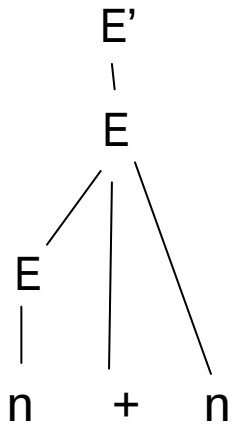
stakk	input
$	t_1 t_2 t_3 t_4 t_5 t_6 $t_7$$
$ t_1	t_2 t_3 t_4 t_5 t_6 t_7 $
$ t_1 t_2	t_3 t_4 t_5 t_6 t_7 $
$ t_1 t_2 t_3	t_4 t_5 t_6 t_7 $
$ t_1 B	t_4 t_5 t_6 t_7 $
$ A	t_4 t_5 t_6 t_7 $
$ A t_4	t_5 t_6 $t_7$$
$ A t_4 t_5	t_6 t_7 $
$ A A	t_6 t_7 $
$ A A t_6	t_7 $
$ A B	t_7 $
$ A B t_7	$
$ S	$
$ S'	$

- Stakk + input: Høyre-avledninger i omvendt rekkefølge
- Får to typer steg:
  - Reduksjon (på toppen av stakken med A → α)
  - Lesing ("skift") av input til stakken
- Dersom man kjenner syntakstreet, er det lett å angi de rette stegene.
- MEN: Hvordan gjøre dette underveis uten å kjenne resten av input ??

Husk at vi skal nå "redusere" input (bottom up) til startsymbolet S', IKKE produsere input fra startesymbolet (slik vi gjorde "top-down" i kap 4)

Eksempel på LR-parsering

$$E' \rightarrow E$$
$$E \rightarrow E + \mathbf{n} \mid \mathbf{n}$$

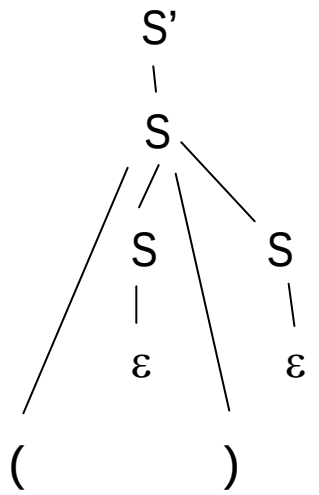


Boka: (Stakk + input) utgjør et stadium i en høyre-avledning. Den neste reduksjonen som skal gjøres, kalles situasjonens ”handle” (håndtak).

	Parsing stack	Input	Action
1	$	**n** + **n** $	shift
2	$ **n**	+ **n** $	reduce $E \rightarrow \mathbf{n}$
3	$ E	+ **n** $	shift
4	$ E +	**n** $	shift
5	$ E + **n**	$	reduce $E \rightarrow E + \mathbf{n}$
6	$ E	$	reduce $E' \rightarrow E$
7	$ E'	$	accept

Eksempel på LR-parsering

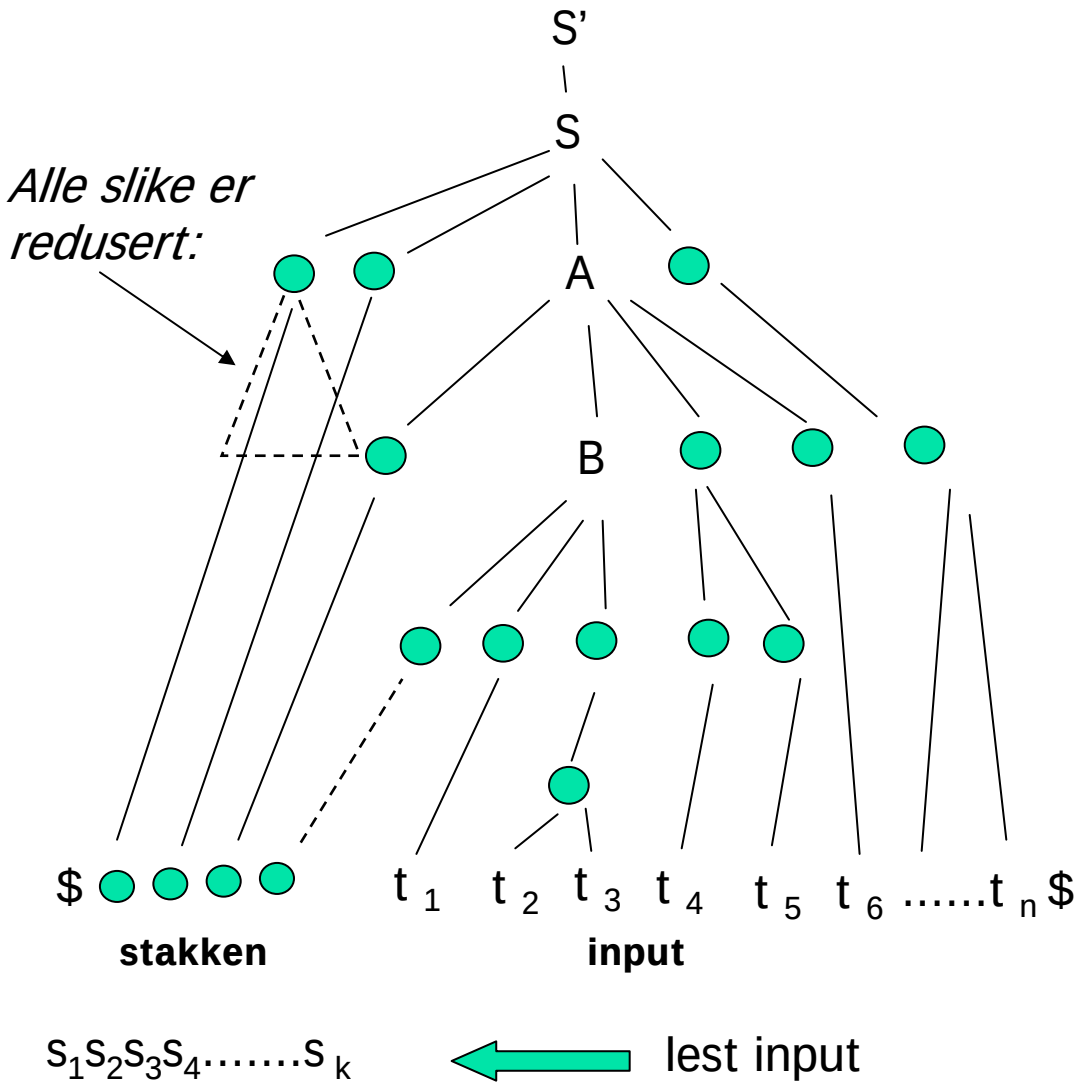
$S' \rightarrow S$
$S \rightarrow (\ S \ ) \quad S \mid \varepsilon$



NB: S → ε
blir litt ”rart”

	Parsing stack	Input	Action
1	$	( ) $	shift
2	$ (	) $	reduce $S \rightarrow \varepsilon$
3	$ (S	) $	shift
4	$ (S)	$	reduce $S \rightarrow \varepsilon$
5	$ (S) S	$	reduce $S \rightarrow (\ S \ ) \ S$
6	$ S	$	reduce $S' \rightarrow S$
7	$ S'	$	accept

Typisk situasjon under LR-parsering



Gitt entydig grammatikk G.

Denne behandles som følger for å lage en LR-parser:

(Ikke alt er like tydelig beskrevet i Boka, men bare boka er pensum)

- Vi ser på de mulig stakker som kan opptre, og ser disse som strenger over alfabetet {terminaler, ikke-terminaler}. Vi ser altså på:

  {S | S utgjør stakken på ett eller annet stadium i LR-parsering av en setning i L(G)}
- Dette språket viser seg å være regulært, og kan beskrives av en NFA der:
  - Tilstandene angis av ”itemer” av formen: $A \rightarrow XY.Z$
  - Kantene kan beskrives nokså greit (kommer snart)
- Denne NFA-en gjør vi om til en DFA på vanlig måte (subset construction fra kap.2)
- Tilstandene i denne DFA-en vil altså være **mengder av itemer**
- **LR-parseringens hovedprinsipp (uformelt):**
  - Gitt en lovlig stakk og anta at den fører DFA-en til en (aksepterende!) tilstand T:
  - Da vil mengden av *itemer* i T angi de mulige ”lokale forhold” vi, i parseringen nå, kan ha ut fra dette stakk-innholdet (det kan være flere muligheter/valg, siden vi ikke har tatt hensyn til resten av input).

Items: Alle produksjonene i BNF-grammatikken omarbeides

- I stedenfor :

  $A \rightarrow \alpha X \beta$

- Legger vi inn nye produksjoner med punktum på alle plasser (punktumet er et Meta-symbol):

  $A \rightarrow \bullet \alpha X \beta$

  $A \rightarrow \alpha \bullet X \beta$

  $A \rightarrow \alpha X \bullet \beta$

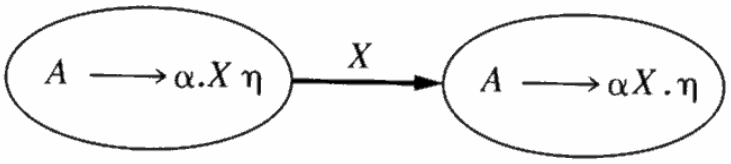
  $A \rightarrow \alpha X \beta \bullet$

- Punktumet sier grovt sett hvilken del av den originale produksjonen vi prøver å jobbe med
  - Det til venstre for punktum er gjenkjent fra input, enten bare lest eller at deler av det er redusert til en ikke-terminal
  - Det til høyre for punktum har vi enda ikke sett/lest

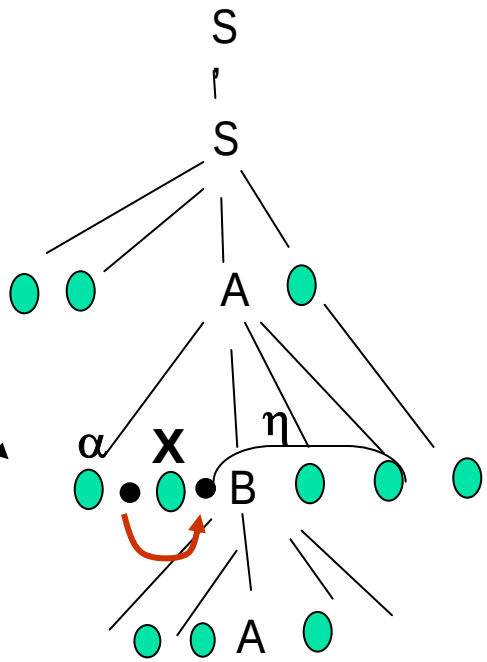
*Disse produksjonene med punktum kalles **items***

Kantene i NFA-en

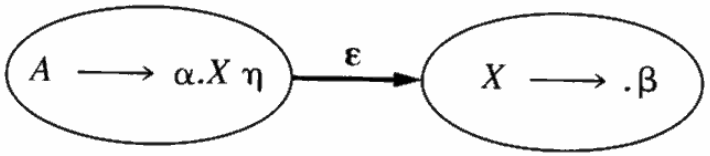
Dette er ikke fullt forklart i boka



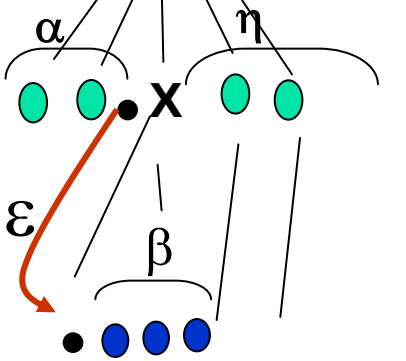
tilsvarer



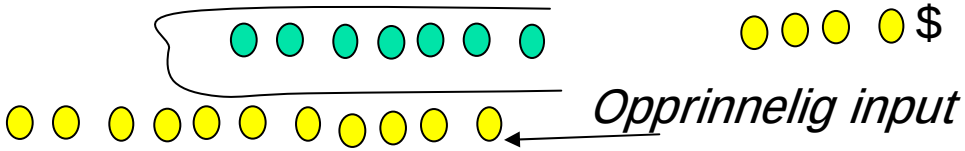
Gitt en grammatikk G og en situasjon under passering av en setning s i L(G).



tilsvarer



Stakk



Opprinnelig input

Eksempel på NFA

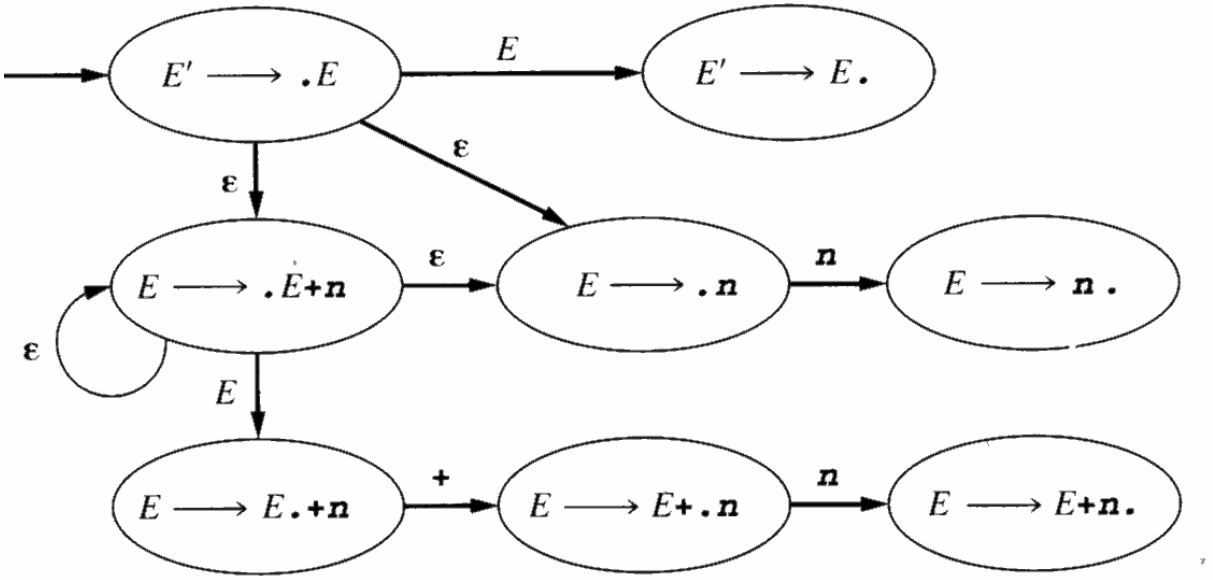
E ' → E

E → E + n

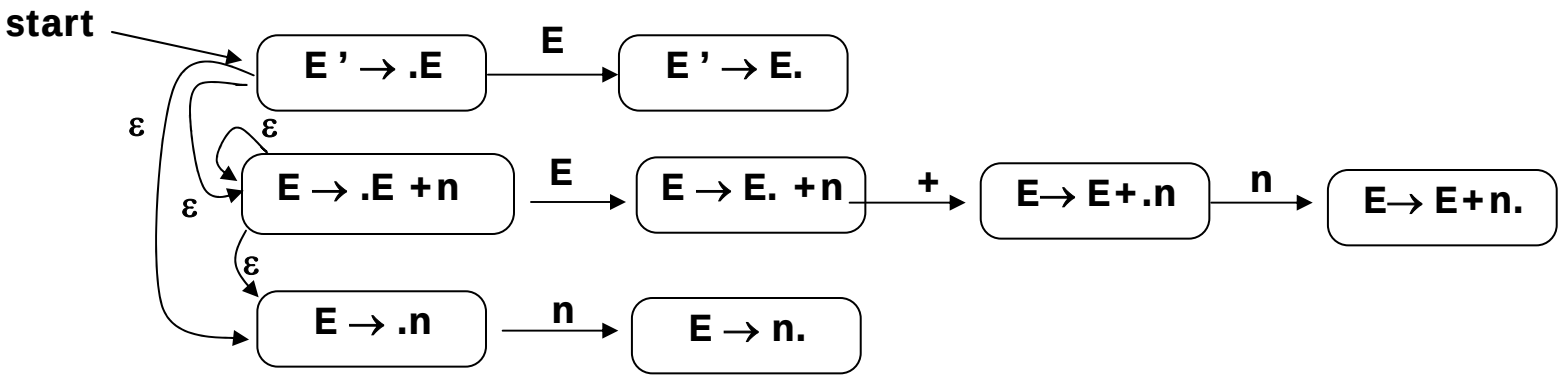
E → n

$E' \rightarrow .E$
$E' \rightarrow E.$
$E \rightarrow .E + n$
$E \rightarrow E.+ n$
$E \rightarrow E + .n$
$E \rightarrow E + n.$
$E \rightarrow .n$
$E \rightarrow n.$

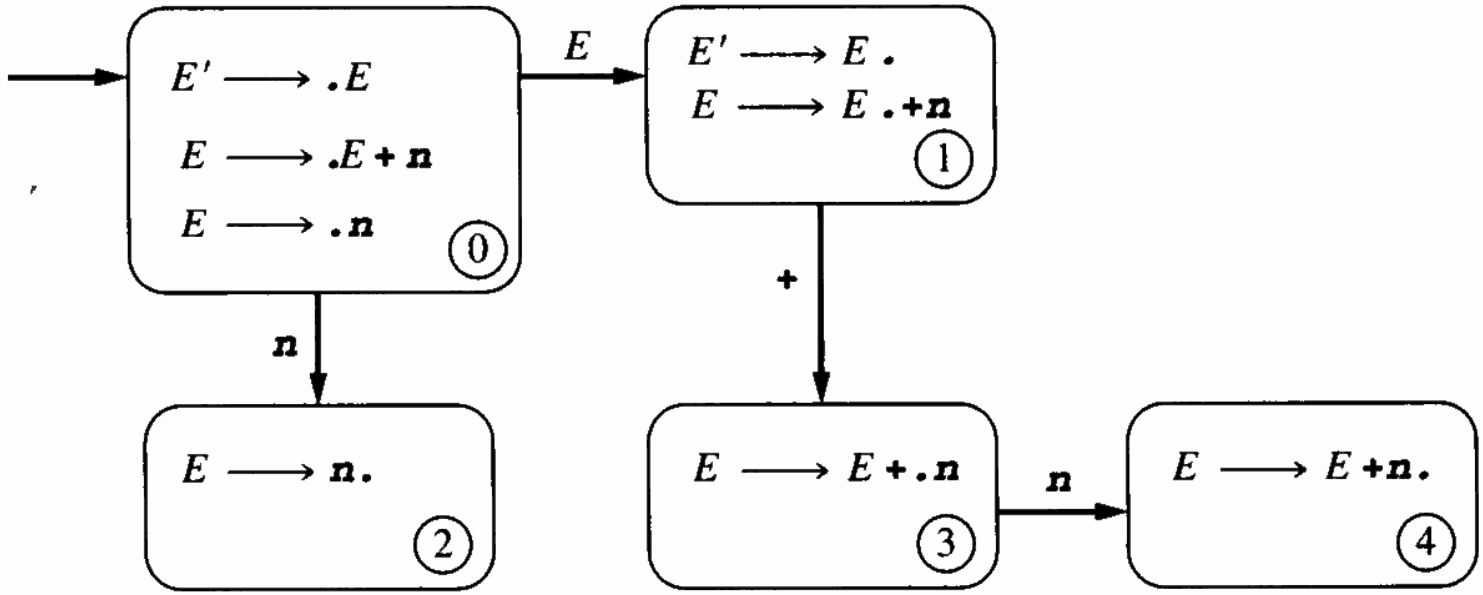
Itemer (LR(0) - itemer)



LR(0) – NFA (litt mer systematisk enn boka)



LR(0) - DFA



Oppgave

Levert ut torsdag 13. febr. 2007

Gitt grammatikk, lag skanner, parser, AST og interpret for språket:

Grammatikk: exp → exp op term | term
term → number | (exp)
op → + | - | *

Bergning gjøres slavisk fra venstre mot høyre (venstreassosiativt), om ikke parentes

- Skriv om grammatikken til en ren BNF-grammatik uten venstre-rekursjon
- Foreta evt. venstre-faktorisering
- Finn First og Follow-mengdene
- Lag M[N,T] – tabellen
- Lag i f.eks Java:
  - En scanner for språket
    - som kan lese et program token for token
    - levere selve leksemet (som tekst)
    - bestemme token-type: NUMB, OPRT, LPAR, RPAR og DOLR ($)
    - bruk f.eks pakken easyIO fra INF1000
  - En parser for språket
    - bygger opp et syntakstre
    - skriver ut treet: først på prefiks form, så på postfiks form
  - En interpret
    - går gjennom treet og beregner verdien til uttrykket

To enkle eksempler på utskrift av kjøring:

Input: 1 * (2 – 3) + 5
Prefiks form: + * 1 – 2 3
Postfiks form: 1 2 3 - * 5 +
Verdi: 4

Input: 1 + 2 + 3 * 4 * 5
Prefiks form: * * + + 1 2 3 4 5
Postfiks form 1 2 + 3 + 4 * 5 *
Verdi: 120

En ikke-venstre-rekursiv BNF grammatikk,

Opprinnelig:

exp → *exp op term* | *term*
term → *number* | (*exp*)
op → + | - | *

Uten venstre-rekursjon:

exp → *term exp2*
exp2 → *op term exp2* | ε
term → number | (*exp*)
op → + | - | *

Her intet å faktorisere!

	First
exp	number, (
exp2	+, -, *, ε
term	number, (
op	+, -, *

	Follow
exp	$,)
exp2	$,)
term	$,), +, -, *
op	number, (

Parseringstabell M[N,T] – er den LL(1)?

exp → *term exp2*

exp2 → *op term exp2* | ε

term → number | (*exp*)

op → + | - | *

	First
exp	number, (
exp2	+, -, *, ε
term	number, (
op	+, -, *

	Follow
exp	$,)
exp2	$,)
term	$,), +, -, *
op	number, (

	number	+	-	*	(	)	$
exp	*exp* → *term exp2*				*exp* → *term exp2*		
exp2		*exp2* → *op term exp2*	*exp2* → *op term exp2*	*exp2* → *op term exp2*		*exp2* → ε	*exp2* → ε
term	*term* → *number*				*term* → (*exp*)		
op		*op* → +	*op* → -	*op* → *			

Den **er** altså LL(1)! (og dermed vet vi også at den er entydig!)

Scanner

(Veldig lite avansert! Litt ”public”-antydninger her, men ikke på senere foiler)

```
import easyIO.*;

class Scanner {
    In infile;   boolean eof = false;
    public int NUMB = 1, OPRT = 2; LPAR = 3; RPAR= 4; DOLR = 5;
    public int cT;  String cLex;     // Her kan brukeren lese “current token” og “current leksem”

    public Scanner(String filnavn) {
        infile = new In(filnavn);
        readNext();  // Hent inn første token
    }

    public readNext() {              // Setter nye verdier i “cT” of “cLex”
        if (eof || infile.endOfFile()) {
           cLex = “”; cT=DOLR; eof = true;
        } else {
            cLex= infile.inWord();   // Forlanger altså blanke/linjeskift mellom symbolene i input.
            char c = cLex.charAt(0);
            if (Character.isDigit(c) ) cT = NUMBER;   // Sjekker ikke fullt ut at det er et tall
            else if ( c== ‘+’ || c==‘-’ || c==‘*‘ ) {cT = OPRT;}
            else if ( c== ‘(‘ ) {cT = LPAR;}
            else if ( c== ‘)‘ ) {cT = LPAR;}
            else {error(“...”);}
        }
    }
}
```

Parser uten trebygging

```
class Parser {   Scanner s;
  Parser(String filnavn) {s = new Skanner(filnavn); }

  expression() { exp(); if (s.cT != s.DOLR) error("...");}        // Hovedmetode

  exp() {
    if ( s.cT == s.NUMB or cT == s.LPAR ) { term(); exp2(); }
    else { error("......");}
  }

  exp2() {
    if (s.cT == s.OPRT ) { op(); term(); exp2();}          // Sa feil på forelesningen. Skal ikke være "readNext" her
    else if (s.cT == s.RPAR or cT == s.DOLR) { /* nothing */ }
    else { error(" ..."); }
  }

  term() { Exp et;
     if (s.cT == NUMB ) {  s.readNext(); }
     else if (s.cT == s.LPAR) {
        s.readNext();  exp();
        if (s.cT == s.RPAR) s.readNext() else error("...");
     }
  }

  op(){
    if (s.cT == s.OPRT) {
       s.readNext();
    } else {error(" ..."); }
  }
} // class Parser
```

// Denne "readNext" ordner biffen!

	number	+	-	*	(	)	$
exp	*exp* → *term exp2*				*exp* → *term exp2*		
exp2		*exp2* → *op term exp2*	*exp2* → *op term exp2*	*exp2* → *op term exp2*		*exp2* → ε	*exp2* → ε
term	*term* → *number*				*term* → (*exp*)		
op		*op* → +	*op* → -	*op* → *			

Trenode-klassene

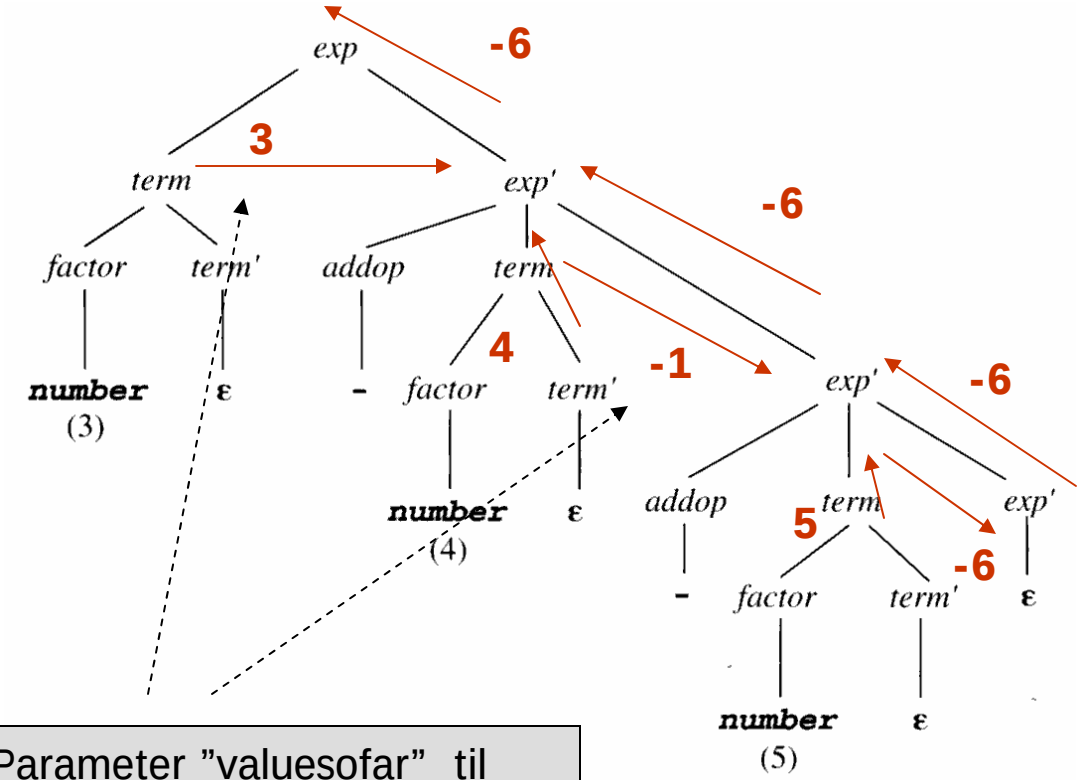
```
class Exp{
   abstract void prefix();
   abstract void postfix();
   abstract int value();
}

class OpNode extends Exp {
   char oprt;   Exp lt, rt;

   OpNode(char oprt, Exp lt, Exp rt) { this.oprt = oprt; this.lt = lt; this.re = rt; }

   void prefix() { print(" " + oprt); lt.prefix(); rt.prefix();}

   void postfix() { lt.postfix(); rt.postfix(); print(" " + oprt);}

   int value() { int lv= lt.value(); int rv = rt.value(); int v;
      switch (oprt) {  '+': v= lv + rv; break;    '-': v = lv - rv; break;   '*': v= lv * rv; break;  }
      return v;
   }
}

class NumNode extends Exp {
   int value;

   NumNode(int value) { this.value = value;}

   void prefix() { print(" " + value);}

   void postfix() { print(" " + value);}

   int value() { return value;}
}
```

Tidligere Foil. Vi må gjøre noe liknende som dette:

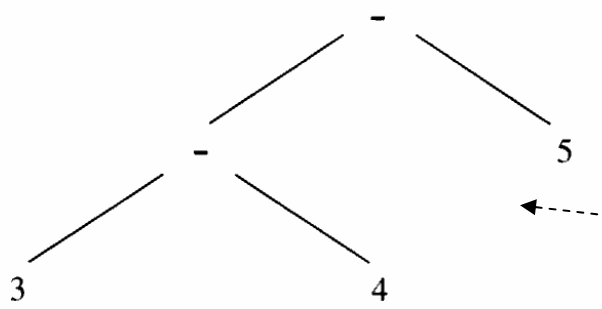
Rec.decent etter tradisjonell fjerning av venstre-rekursjon (treet er nå høyre assosiativt istedenfor venstre). Det må korrigeres for.

Lage tre eller beregne verdi : 3 – 4 – 5

$exp \rightarrow term\ exp'$
$exp' \rightarrow addop\ term\ exp' \mid \varepsilon$
$addop \rightarrow \mathbf{+} \mid \mathbf{-}$
$term \rightarrow factor\ term'$
$term' \rightarrow mulop\ factor\ term' \mid \varepsilon$
$mulop \rightarrow \mathbf{*}$
$factor \rightarrow \mathbf{(}\ exp\ \mathbf{)} \mid \mathbf{number}$



Det abstrakte syntakstreet vi ønsker å lage:



Parameter ”valuesofar” til prosedyren ”exp”

For trebygging ville den være: ”rootOfTreeSoFar”

Antar implisitt adgang til Scanneren, og har fjernet alle ”s.”. Med trebygging

Gammelt program

```
class Parser { ...
  expression() { exp(); if (cT != DOLR) error(”...”);}

  exp() {
    if (ct == NUMB or cT == LPAR){ term(); exp2();}
    else { error(”.....”);}
  }

  exp2() {
    if (cT == OPRT ) { op(); term(); exp2();}
    else if (cT == RPAR or cT == DOLR) { }
    else { error(” ...”); }
  }

  term() {
     if (cT == NUMB ) {  readNext(); }
     else if (cT == LPAR) {
        readNext();  exp();
        if (cT==RPAR) {readNext();} else {error(”...”);}
     }
  }

  op(){
    if (cT == OPRT) {
       readNext();
    } else {error(” ...”); }
  }
} // class Parser
```

```
class Parser { ...
  Exp expression(){ Exp et;  et=exp(); if (cT != DOLR) error(”...”); return et;}

  Exp exp() { Exp et, etsf;
    if ( cT == NUMB or cT == LPAR ) { etsf = term(); et=exp2(etsf);}
    else { error(”.....”);}
    return et;
  }

  Exp exp2(etin) { Exp et, etsf; char oprt;
    if (cT == OPRT ) { oprt = op(); et= term();
                       etsf= new OpNode(oprt, etin, et);  et = exp2(etsf);}
    else if (cT == RPAR or cT == DOLR) { et = etin }
    else { error(” ...”);}
    return et;
  }

  Exp term() { Exp et;
     if (cT == NUMB ) {et = new NumNode(cLex.intValue); readNext(); }
     else if (cT == LPAR) {
        readNext(); et = exp();
        if (cT == RPAR) {readNext();} else {error(”...”);}
     } else {error(”...”);}
     return et;
  }

  char op(){   char oprt;
     if (cT == OPRT) {
        oprt = cLex.charAt(0); readNext();
     } else {error(” ...”); }
     return oprt;
  }
} // class Parser
```

Litt frekk

```
class Parser { ...
  Exp expression() { Exp et;  et=exp(); if (cT != DOLR) error("..."); return et;}

  Exp exp(){
       Exp et; char oprt; Exp xet;
       et= term();
       while (cT == OPRT) {
          oprt = op(); readNext(); xet = term();
          et= new OpNode(oprt, et, xet);
       }
       return et;
  }

   Exp term() { Exp et;
       if (cT == NUMB ) {et = new NumNode(cLex.value); readNext(); }
       else if (cT == LPAR) {
          readNext(); et = exp();
          if (cT == RPAR) readNext() else error("...");
       }
       return et;
   }

   char op(){
       if (cT == OPRT) {
          oprt = cLex.charAt(0); readNext();
       } else { error(" ..."); }
       return oprt;
   }
} // class Parser
```

Parsering etter EBNF

Mye likt det på side 150/51

Opprinnelig:

exp → *exp op term* | *term*

term → *number* | (*exp*)

op → + | - | *

Gjort om til EBNF:

exp → *term* { *op term* }

term → number | (*exp*)

op → + | - | *

Hovedprogrammet

```
import easyIO.*;

class Scanner { ... }

class Parser { ... }

class Exp{ ... }
class OpNode extends Exp { ... }
class NumNode extends Exp { ... }

class  Compiler {
   Parser parser;

   public static void main(String[] args) {
      parser = new Parser(args[0]);       // Parser lager scanner selv
      Exp tree = parser.expression();
      print(”Prefix: ”); tree.prefix;
      print(”Postfix: ”); tree.postfix;
      print(”Verdi: ” + tree.value);
   }
}
```