Kap.4 del 2
Top Down Parsering
INF5110 – v2006

Arne Maus
Ifi, UiO

LL(1) –tabell for uttrykks-grammatikk

Har fjernet venstre-rekursjon:

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

First(exp) = { (, **number**}	Follow(exp) = {$,) }
First(exp') = {+, -, ε}	Follow(exp') = {$,) }
First($addop$) = {+, -}	Follow($addop$) = { (, **number**}
First($term$) = { (, **number**}	Follow($term$) = {$,), +, -}
First($term'$) = {*, ε}	Follow($term'$) = {$,), +, -}
First($mulop$) = {*}	Follow($mulop$) = { (, **number**}
First($factor$) = { (, **number**}	Follow($factor$) = {$,), +, -, *}

M[N, T]	(	**number**	)	+	-	*	$
exp	$exp \rightarrow term\ exp'$	$exp \rightarrow term\ exp'$					
exp'			$exp' \rightarrow \varepsilon$	$exp' \rightarrow addop\ term\ exp'$	$exp' \rightarrow addop\ term\ exp'$		$exp' \rightarrow \varepsilon$
addop				$addop \rightarrow$ +	$addop \rightarrow$ -		
term	$term \rightarrow factor\ term'$	$term \rightarrow factor\ term'$					
term'			$term' \rightarrow \varepsilon$	$term' \rightarrow \varepsilon$	$term' \rightarrow \varepsilon$	$term' \rightarrow mulop\ factor\ term'$	$term' \rightarrow \varepsilon$
mulop						$mulop \rightarrow$ *	
factor	$factor \rightarrow$ (*exp*)	$factor \rightarrow$ **number**					

Alternativ def. av LL(1) grammatikker

Sier at alle alternativer for A skal ha disjunkte startmengder, og da høyst ett utnullbart alternativ

A grammar in BNF is **LL(1)** if the following conditions are satisfied.

1. For every production $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$, $\text{First}(\alpha_i) \cap \text{First}(\alpha_j)$ is empty for all i and j, $1 \leq i, j \leq n$, $i \neq j$.
2. For every nonterminal A such that First(A) contains ε, First(A) ∩ Follow(A) is empty.

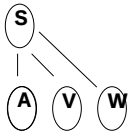
Dersom A er utnullbar, så må First(A) og Follow(A) være disjunkte.

At dette kravet er ekvivalent med det opprinnelige LL(1)-kravet krever et lite bevis.

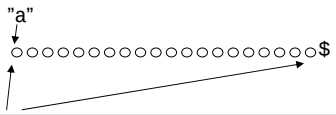
Kap 4.2: LL(1) –parsering med eksplisitt stakk

- Dette kalles i boka bare ”LL(1)-parsering”
- Gjør logisk sett nøyaktig det samme som rek. desc. parsering for ren BNF.
- Bruker LL(1)-tabellen M[T,N] til å styre parseringen

Start-situasjonen: (S er start-symbolet)



input:



hele input, med ”a” som første token

Anta at a ∈ First(S) og at M[S,a] = ”S → AVW”, da bruker vi denne produksjonen (se neste foil) ved å erstatte S på stakken med AVW

Stakk

S $

Starter med S og slutt-symbolet på stakken (End-Of-File EOF).

På stakken har vi det vi kke er ”ferdig med”

La X være på toppen av stakken og ”a” er nåværende token (neste på input)

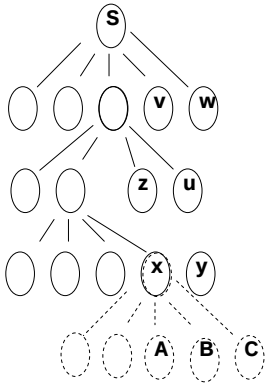
Om X er en terminal:

Da må X == a. Fjern X (pop) og les a

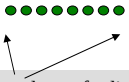
Om X ikke-terminal:

Velg aksjon M[X,a]. Om ikke ”error”, så fjern X (pop X) La M[X,a] være ”X-> ABC” og legg (Push) ABC på stakken (N.B. vi har ikke lest ”a” nå)

Normal-situasjonen: (S er start-symbolet)

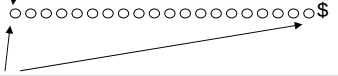


input:



analysert ferdig og funnet OK

”a”



resten av input, med ”a” først (token)

Stakk

XYZUVW $

Dette er en setningsform som oppstår under venstre-avledning fra startesymbolet til input-setningen.

Utførelse av LL(1)-persering med tabell M[N,T] og eksplisitt stakk . Merk: noen feil må rettes.

```
(* assumes $ marks the bottom of the stack and the end of the input *)
push the start symbol onto the top of the parsing stack ;
while the top of the parsing stack ≠ $ and the next input token ≠ $ do
  if the top of the parsing stack is terminal a
       and the next input token = a
  then (* match *)
    pop the parsing stack ;
    advance the input ;
  else if the top of the parsing is nonterminal A
       and the next input token is terminal a
       and parsing table entry M[A, a] contains
            production A → X1X2 . . . Xn
  then (* generate *)
    pop the parsing stack ;
    for i := n downto 1 do
      push Xi onto the parsing stack ;
  else error ;
if the top of the parsing stack = $
       and the next input token = $
then accept
else error ;
```

Det ville stoppe for tidlig (se eksempler)

”top” er terminal

”top” er ikke-terminal

Push baklengs på stakken (slik at første symbol kommer på toppen)

Dette vet vi holder

Aksept hvis vi greide å tolke hele input som S

Hit hvis input er ”for lang”

i

g

M[N, T]	if	other	else	0	1	$
statement	statement → if-stmt	statement → other				
if-stmt	if-stmt → if (exp) statement else-part					
else-part			1 else-part → else statement 2 else-part → ε			else-part → ε
exp				exp → 0	exp → 1	

LL(1)-parsering med en stakk ut fra en tabell som ikke er LL(1).
Må gjøre valg i tvetydige situasjoner

Parsing stack	Input	Action
$ S	i(0)i(1)oeo$	S → I
$ I	i(0)i(1)oeo$	I → i (E) S L
$ L S) E (i	i(0)i(1)oeo$	match
$ L S) E (	(0)i(1)oeo$	match
$ L S) E	0)i(1)oeo$	E → 0
$ L S) 0	0)i(1)oeo$	match
$ L S)	)i(1)oeo$	match
$ L S	i(1)oeo$	S → I
$ L I	i(1)oeo$	I → i (E) S L
$ L L S) E (i	i(1)oeo$	match
$ L L S) E (	(1)oeo$	match
$ L L S) E	1)oeo$	E → 1
$ L L S) 1	1)oeo$	match
$ L L S)	)oeo$	match
$ L L S	oeo$	S → o
$ L L o	oeo$	match
$ L L	eo$	L → e S
$ L S e	eo$	match
$ L S	o$	S → o
$ L o	o$	match
$ L	$	L → ε
$	$	accept

Når kompilatoren oppdager feil

- minstekrav:
  - Tester løpende at programmet er OK, og gir fornuftig feilmelding ved feil (men stopper)
- Vanlig krav ved feil (”error recovery”):
  - Gir fornuftig feilmelding.
  - ”Blar forbi feilen” og fortsetter kompileringen (blar forbi så lite som mulig).
  - Vanligvis vil man slutte å lage maskinkode etter feil (Men noe ”feilrettende” kompilatorer forsøker det – lite brukt)
  - Det er for *syntaksfeil* det er vanskeligst å ta opp tråden etter feil.

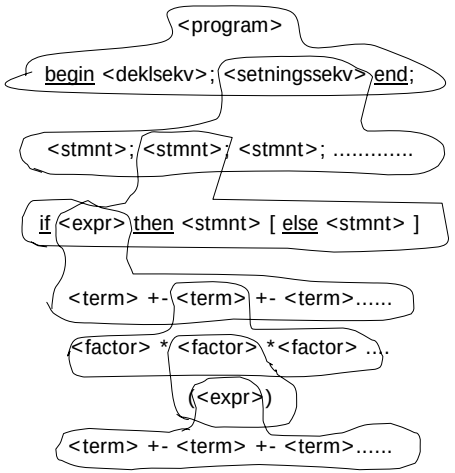
Spesielt viktig ved syntaksfeil:

- Forsøke å unngå feilmeldinger som bare er følgefeil
- Rapportere feil så tidlig som mulig, helst så snart det man har lest *ikke kan forlenges til et riktig program*
- Man må passe på at men ikke blir gående i løkke rapportere feil *uten å lese noe fra input.*

Behandling av Syntaksfeil

ved ”recursive decent” parsering.

Metode:”Panic mode” og synkroniserings-mengde



Synkset:

(stakk eller parameter)

;First(stmt)

then

First (stmt)

else

+,-,(,tall, navn

*, (, tall, navn

)

+,-, (, tall, navn

Syntaksfeil ved ”rec. descent” – 2

Ut fra skissen er det greit å finne:

- hvem som skal ta opp tråden
- ”hvor” denne skal gjøre av det

Algoritme:

For hvert input-symbol framover:

Let gjennom stakken topp → bunn

Om finnes:

Den” tilsvarende proc” skal ta opp tråden

Den vet selv hvor den skal fortsette, med dette symbolet

Om input-symbol ikke er i stakken:

- gå til neste input-symbol

Det som ikke er greit, er å programmere dette uten at den vakre strukturen ved ”rec. descent” blir helt ødelagt.

Uttrykksprosedyrer ved "error recovery"

```
procedure exp ( synchset ) ;
begin
  checkinput ( { (, number }, synchset ) ;
  if not ( token in synchset ) then
    term ( synchset ) ;
    while token = + or token = - do
      match (token) ;
      term ( synchset ) ;
    end while ;
    checkinput ( synchset, { (, number }) ;
 end if;
end exp ;

procedure factor ( synchset ) ;
begin
 checkinput ( { (, number }, synchset ) ;
 if not ( token in synchset ) then
  case token of
  ( : match( ( ) ;
      exp ( { ) } ) ;
      match( ) ) ;
  number :
      match(number) ;
  else error ;
  end case ;
  checkinput ( synchset, { (, number }) ;
 end if ;
end factor ;
```

"error"_skriver feilmelding og returnerer

Bruker parameter, ikke stakk

Prosedyrene må selv ta opp tråden riktig når de får igjen kontrollen:

match(t) er som før:

- tester input mot t
- kaller eventuelt "error"
- kaller ikke "scanto(...)

```
procedure scanto ( synchset ) ;
begin
  while not ( token in synchset ∪ { $ }) do
    getToken ;
end scanto ;

procedure checkinput ( firstset, followset ) ;
begin
  if not ( token in firstset ) then
    error ;
    scanto ( firstset ∪ followset ) ;
  end if ;
end;
```

Pensum i kap. 4

blir lagt ut på hjemmesida

Hele kapittelet, med følgende modifikasjoner:

- side 159-160 : Ikke detaljene i "Case 3", men man skal vite at en slik algoritme finnes.
- Ikke avsnitt 4.2.4 side 166-167
- Sidene 168 og 173: Vi bruker litt mer intuitive definisjoner på First og Follow (se foilene, men også diskusjonen på punktene 1. og 2. øverst på s. 178 der koblingen til de intuitive definisjonene er gjort)
- Ikke avsnittene 4.5.2 og 4.5.3 side 186 – 189

Oppgave: grammatikk, parser (+ skanner og interpret) for språket:
exp →exp op number | number
op → + | -

1. Skriv om grammatikken til en ren BNF-grammatik uten venstre-rekursjon
2. Foreta evt. venstre-faktorisering
3. Finn First og Follow-mengdene
4. Lag M[N,T] – tabellen
5. Lag i f.eks Java:
   1. En scanner for språket
      1. som kan lese språket token for token
         bestemme type + evt. verdi : NUMBER, ADD eller SUBTR
         (bruk f.eks pakken easyIO fra INF1000)
   2. En parser for språket
      1. bygger opp et syntakstre
      2. skriver ut treet på prefix form
   3. En interpret
      1. går gjennom treet og beregner verdien til uttrykket

To enkle eksempler på utskrift av kjøring:
1 – 2 og: 3 – 4- 5

```
>java Tre  12.txt
Test av setning:1 - 2
(SUBTR:(NUMBER:1),(NUMBER:2))

Evaluering av venstre-tre av: 1 - 2 er: -1

> java Tre  345.txt
Test av setning: 3 - 4 - 5
(SUBTR:(SUBTR:(NUMBER:3),(NUMBER:4)),(NUMBER:5))

Evaluering av venstre-tre av:  3 - 4 - 5 er: -6
```

```
java Tre  1-6.txt
Test av setning:1 - 2 - 3 - 4 - 5 - 6 + 6 + 5 + 4 + 3 + 2

(ADD:(ADD:(ADD:(ADD:(ADD:(SUBTR:(SUBTR:(SUBTR:(SUBTR:(SUBTR:(NUMBE
R:1),(NUMBER:2)),(NUMBER:3)),(NUMBER:4)),(NUMBER:5)),(NUMBER:6)),(NUM
BER:6)),(NUMBER:5)),(NUMBER:4)),(NUMBER:3)),(NUMBER:2))

Evaluering av venstre-tre av: 1 - 2 - 3 - 4 - 5 - 6 + 6 + 5 + 4 + 3 + 2 er: 1
```